

MiniPascal

Language

Manual

Table of Contents

Unit/Section	Topic	Page
	Preface	18.14
	Notation	18.14
Unit I	Using MiniPascal as a File Format	
	Introduction	18.15
1-1	Beginning Steps	
	Importing text files.	18.1
	Exporting text files.	18.17
1-2	Getting Started	
	An introduction to statements and standard procedures.	18.19
	Use of the semicolon.	18.20
	The absolute, relative and distance-angle methods of point designation.	18.20
Unit II	Variables and Data Types in MiniPascal	
	Introduction	18.27
2-1	Data Types	
	Simple Data Types	18.27
	Scalar types	18.27
	The integer type	18.27
	The longint type	18.27
	The boolean type	18.27
	The char type	18.28
	Real types	18.28
	String types	18.28
	Handle types	18.28
	Structured Types	18.28
	Array types	18.28
	Vector types	18.28
	Text file types	18.28

Unit III Using MiniPascal as a Programming Language

Introduction	18.31
3-1 Symbols	
Special symbols and reserved words	18.33
Identifiers	18.34
Numbers	18.35
Labels	18.35
Quoted string and character constants	18.35
Delimiters	18.36
3-2 Expressions	
Operators	18.37
Arithmetic operators	18.37
Boolean operators	18.41
Relational operators	18.46
Comparing numbers	18.46
Comparing booleans	18.47
Comparing strings	18.47
Function calls	18.49
3-3 Variables and Constants	
Variable declarations	18.51
Constant declarations	18.52
3-4 Statements	
Assignment statements	18.53
Procedure statements	18.56
Conditional statements	18.56
IF statements	18.57
Compound statements	18.58
Nested IF statements	18.60
Repetition statements	18.61
FOR statements	18.62
REPEAT statements	18.64
WHILE statements	18.65
Control statements	18.66
GOTO statements	18.66

3-5	Procedures and Functions	
	Procedure declarations	18.68
	Function declarations	18.70
	Local and global identifiers	18.72
	Blocks and scope	18.72
	Parameters	18.74
	Value parameters	18.74
	Variable parameters	18.74
	Parameter list compatibility	18.74

Unit IV Importing and Exporting Information in MiniPascal

	Introduction	18.77
--	---------------------------	-------

4-1 Using Text Files in MiniPascal

	Text files	18.77
	File variables	18.77
	Opening a file for writing	18.77
	Closing a write file	18.78
	Opening a file for reading	18.78
	Closing a read file	18.78

Unit V MiniPascal Pre-defined Routines

	Introduction	18.79
--	---------------------------	-------

5-1 Selection Routines

	Procedure SelectAll	18.79
	Procedure DSelectAll	18.80

5-2 Creation Routines

	Procedure Arc	18.81
	Procedure DimArcText	18.82
	Procedure DimText	18.82
	Procedure Layer	18.83
	Procedure Line	18.84
	Procedure LineTo	18.84
	Procedure Locus	18.85
	Procedure Move	18.86
	Procedure MoveTo	18.86
	Procedure Oval	18.87
	Procedure PenLoc	18.88
	Procedure Poly	18.88
	Procedure Poly3D	18.89
	Procedure Rect	18.91

Procedure RRect	18.92
Procedure BeginGroup	18.94
Procedure EndGroup	18.94
Procedure Group	18.98
Procedure Ungroup	18.98
Procedure SprdSheet	18.99
Procedure SprdAlign	18.100
Procedure SprdBorder	18.101
Procedure SprdFormat	18.101
Procedure SprdWidth	18.103
Procedure LoadCell	18.104
Procedure BeginSym	18.105
Procedure EndSym	18.105
Procedure Symbol	18.105
Procedure BeginFolder	18.106
Procedure EndFolder	18.106
Procedure TextOrigin	18.107
Procedure BeginText	18.107
Procedure EndText	18.107
Procedure BeginMesh	18.107
Procedure EndMesh	18.107
Procedure BeginXtrd	18.109
Procedure EndXtrd	18.109
Procedure BeginMXtrd	18.110
Procedure EndMXtrd	18.110
Procedure BeginSweep	18.112
Procedure EndSweep	18.115
Procedure BeginPoly	18.115
Procedure AddPoint	18.115
Procedure EndPoly	18.115
Procedure Message	18.116
Procedure ClrMessage	18.116

5-3 Attribute Routines

Procedure ArrowHead	18.117
Procedure ArrowSize	18.118
Procedure ClosePoly	18.118
Procedure OpenPoly	18.118
Procedure CopyMode	18.119
Procedure NameObject	18.120
Procedure NameClass	18.120
Procedure FillPat	18.121
Procedure FillFore	18.122
Procedure FillBack	18.122
Procedure PenFore	18.122
Procedure PenBack	18.122

18.122	Procedure LckObjs	18.122
18.122	Procedure UnlckObjs	18.122
18.122	Procedure PenPat	18.122
18.123	Procedure PenSize	18.123
18.124	Procedure Smooth	18.124
18.126	Procedure ShowLayer	18.126
18.126	Procedure HideLayer	18.126
18.126	Procedure GrayLayer	18.126
18.126	Procedure ShowClass	18.126
18.126	Procedure HideClass	18.126
18.126	Procedure TextFace	18.126
18.128	Procedure TextFlip	18.128
18.129	Procedure TextFont	18.129
18.129	Procedure TextJust	18.129
18.131	Procedure TextRotate	18.131
18.132	Procedure TextSize	18.132
18.133	Procedure TextSpace	18.133

5-4 Global Routines

18.134	Procedure Absolute	18.134
18.135	Procedure Relative	18.135
18.135	Procedure AngleVar	18.135
18.135	Procedure NoAngleVar	18.135
18.136	Procedure DoubLines	18.136
18.137	Procedure DrwSize	18.137
18.138	Procedure GridLines	18.138
18.138	Procedure PenGrid	18.138
18.139	Procedure Redraw	18.139
18.139	Procedure SetOrigin	18.139
18.140	Procedure SetScale	18.140
18.141	Procedure Snap	18.141
18.141	Procedure SetUnits	18.141
18.141	Procedure Units	18.141

5-5 Alteration Routines

18.144	Procedure DeleteObjs	18.144
18.144	Procedure Duplicate	18.144
18.145	Procedure FlipHor	18.145
18.145	Procedure FlipVer	18.145
18.146	Procedure Forward	18.146
18.146	Procedure Backward	18.146
18.146	Procedure MoveFront	18.146
18.146	Procedure MoveBack	18.146
18.147	Procedure MoveObjs	18.147
18.148	Procedure Move3D	18.148
18.149	Procedure Rotate	18.149

Procedure Rotate3D	18.149
Procedure Scale	18.150
Procedure DelClass	18.151
Procedure DelName	18.151
Procedure VSave	18.151
Procedure VRestore	18.151
Procedure VDelete	18.151

5-6 Utility Routines

Procedure Sysbeep	18.152
Function FndError	18.152
Function Date	18.152
Procedure Wait	18.153

5-7 Dialog Routines

Function AngDialog	18.154
Function DistDialog	18.155
Function IntDialog	18.156
Procedure PtDialog	18.157
Function RealDialog	18.158
Function StrDialog	18.159
Function YNDialog	18.160
Procedure AlrtDialog	18.161
Procedure AngDialog3D	18.161
Procedure PtDialog3D	18.163
Function DidCancel	18.164

5-8 Input and Output Routines

Procedure Open	18.165
Procedure GetFile	18.165
Procedure Read	18.166
Procedure Readln	18.167
Function EOF	18.168
Function EOLN	18.168
Procedure Rewrite	18.168
Procedure PutFile	18.169
Procedure Append	18.169
Procedure Close	18.170
Procedure Write	18.170
Procedure Writeln	18.173
Procedure Tab	18.174
Procedure Space	18.174

5-9 Menu Routines

Procedure ForEachObject	18.175
Procedure Run	18.176
Procedure DoMenu	18.176

5-10 Inquiry Routines

Introduction	18.181
Function CellString	18.189
Function CellValue	18.189
Function Angle	18.190
Function Area	18.190
Function Count	18.190
Function Height	18.191
Function Length	18.191
Function ObjectType	18.192
Function Perim	18.193
Procedure SelectObj	18.193
Procedure DSelectObj	18.193
Function Width	18.194
Function XCenter	18.194
Function YCenter	18.194
Function LeftBound	18.195
Function TopBound	18.195
Function RightBound	18.196
Function BotBound	18.196

Unit VI Standard Pre-defined Routines

Introduction	18.198
--------------------	--------

6-1 Transfer functions

Function Trunc	18.198
Function Round	18.198

6-2 Arithmetic functions

Function Abs	18.199
Function Sqr	18.199
Function Sin	18.199
Function Cos	18.199
Function Exp	18.199
Function Ln	18.199
Function Sqrt	18.200
Function Arctan	18.200
Function ArcSin	18.200
Function ArcCos	18.200
Function Rad2Deg	18.200

	Function Deg2Rad	18.200
6-3	Ordinal Functions	
	Function Ord	18.201
	Function Chr	18.201
6-4	String Procedures and Functions	
	Function Len	18.202
	Function Pos	18.202
	Function Concat	18.202
	Function Copy	18.202
	Procedure Delete	18.202
	Procedure Insert	18.203
	Function Str2Num	18.203
	Function Num2Str	18.203
	Function Num2StrF	18.203
	Procedure UpString	18.203
6-5	Graphic Calculation Routines	
	Function PtlInPoly	18.204
	Function Distance	18.204
	Function PtlInRect	18.204
	Procedure UnionRect	18.204
	Function EqualRect	18.204
	Function EqualPt	18.204
Unit VII	Using Handle Variables	
	Introduction	18.205
7-1	Handle Variables	18.205
7-2	Handle Routines Which Obtain a Connection	
	Function FObject	18.206
	Function LObject	18.206
	Function FActLayer	18.206
	Function FSActLayer	18.207
	Function LSActLayer	18.207
	Function ActSSheet	18.207
	Function GetObject	18.207
	Function FLayer	18.207
	Function LLayer	18.207
	Function ActLayer	18.207
	Function FSOBJect	18.207
	Function GetLayer	18.208
	Function FSymDef	18.208

Function FlnGroup	18.208
Function Fln3D	18.209
Function FlnSymDef	18.208
Function FlnLayer	18.209
Function FlnFolder	18.209
Function PickObject	18.209

7-3 Routines Which Return Object Information

Function GetType	18.211
Function GetName	18.211
Function GetClass	18.211
Procedure GetBBox	18.211
Function GetFPat	18.212
Function GetLW	18.212
Function GetLS	18.212
Function Selected	18.212
Function GetText	18.212
Procedure GetSegPt1	18.213
Procedure GetSegPt2	18.213
Function HasDim	18.213
Function GetDimText	18.213
Function GetSymName	18.213
Function GetSymRot	18.214
Procedure GetLocPt	18.214
Procedure GetRRDiam	18.214
Procedure GetArc	18.214
Function GetVertNum	18.214
Procedure GetPolyPt	18.214
Procedure Get3DCntr	18.214
Procedure Get3DInfo	18.214
Procedure SprdSize	18.214
Function CellHasStr	18.215
Function CellHasNum	18.215
Function GetCellStr	18.215
Function GetCellNum	18.215
Function GetCWidth	18.215
Function GetCAlign	18.215
Function HArea	18.216
Function HPerim	18.216
Function HLength	18.216
Function HWidth	18.216
Function HHeight	18.216
Function HAngle	18.217
Procedure HCenter	18.217

7-5	Routines Which Return Symbol Information	
	Function GetSDName	18.217
7-6	Routines Which Return File Information	
	Function GetFName	18.218
	Function NumLayers	18.218
	Function SymDefNum	18.218
	Function NameNum	18.219
	Function NameList	18.219
	Function ClassNum	18.219
	Function ClassList	18.219
	Procedure GetOrigin	18.219
	Function FArrowHead	18.219
	Function FFillPat	18.219
	Function FPenSize	18.219
	Function FPenPat	18.219
	Procedure FPenFore	18.220
	Procedure FPenBack	18.220
	Procedure FFillFore	18.220
	Procedure FFillBack	18.220
7-7	Routines Which Change a Graphic Object	
	Procedure SetSelect	18.220
	Procedure SetDSelect	18.220
	Procedure SetName	18.220
	Procedure SetClass	18.220
	Procedure SetFPat	18.220
	Procedure SetLW	18.221
	Procedure SetLS	18.221
	Procedure SetBBox	18.221
	Procedure SetPenFore	18.221
	Procedure SetPenBack	18.221
	Procedure SetFillFore	18.221
	Procedure SetFillBack	18.222
	Procedure SetSegPt1	18.222
	Procedure SetSegPt2	18.222
	Procedure SetPolyPt	18.222
	Procedure SetArc	18.222
	Procedure Set3DRot	18.222
	Procedure Set3DInfo	18.222
	Procedure Move3DObj	18.223
	Procedure SelectSS	18.223
	Procedure SetText	18.223

7-8	Routines Which Change File Settings	
	Procedure SetView	18.223
	Procedure SetCursor	18.223
7-9	Handle Traversing Routines	
	Procedure ForEachObj	18.224
	Function NextLayer	18.225
	Function PrevLayer	18.225
	Function NextObj	18.225
	Function PrevObj	18.225
	Function NextSObj	18.225
	Function PrevSObj	18.225
	Function NextDObj	18.225
	Function PrevDObj	18.226
	Function NextSymDef	18.226
	Function PrevSymDef	18.226
Unit VIII	Using Array and Vector Variables	
	Introduction	18.227
8-1	Array Variables	18.227
8-2	Vector Variables	
	Function Norm	18.232
	Function Perp	18.232
	Function Vec2Ang	18.232
	Function Ang2Vec	18.232
	Function UnitVec	18.232
	Procedure Comp	18.232
	Function AngBVec	18.232
Unit IX	Interactive Routines	
	Introduction	18.233
9-1	Specific Event Routines	
	Procedure GetRect	18.233
	Procedure GetLine	18.233
	Procedure GetPt	18.234
	Procedure GetPtL	18.234
	Procedure GetKeyDown	18.235

9-2	Event Monitoring Routines	
	Function MouseDown	18.236
	Function KeyDown	18.236
	Function AutoKey	18.237
	Procedure GetMouse	18.238
	Function Option	18.238
	Function CapsLock	18.238
	Function Shift	18.238
	Function Command	18.238
 9-3	 Custom Dialog Routines	
	Procedure BeginDialog	18.239
	Procedure EndDialog	18.240
	Procedure AddButton	18.240
	Procedure AddField	18.241
	Procedure GetDialog	18.241
	Procedure ClrDialog	18.242
	Procedure DialogEvent	18.242
	Function GetField	18.242
	Procedure SetField	18.242
	Procedure SelField	18.242
	Procedure SetItem	18.242
	Function ItemSel	18.243
	Function ValNumStr	18.243
	Function ValAngStr	18.243
	Procedure GetScreen	18.243
 10-1	 Database & Worksheet Routines	
	Procedure NewField	18.244
	Procedure SetRecord	18.244
	Procedure SetField	18.244
	Function Eval	18.244
	Function EvalStr	18.245
 10-2	 Additional Routines	
	Function GetFont	18.245
	Function GetSize	18.245
	Function GetStyle	18.245
	Function GetLVis	18.246
	Procedure HMove	18.246

Procedure SetActSymbol	18.24
Function ActSymDef	18.246
Function LNewObj	18.246
Procedure SetTool	18.246
Procedure SetConstrain	18.247

Glossary	Glossary of Terms	18.248
Appendix A	Solutions to Exercises	18.251
Appendix B	Reserved Words and the ASCII Character Set	18.253
Index	Index of Technical Terms	18.255
Reference	MiniPascal Quick Reference Section	18.266

Preface Welcome to the MiniPascal language manual. MiniPascal is a programming language which is a component of the graphics software MiniCad+ by Graphsoft, Inc. The language creates a flexible working environment within MiniCad+ by allowing the user two ways to implement subroutines. First, it allows him to execute pre-defined subroutines. Secondly, it allows him to create his own subroutines and execute them from within the application.

The manual has been written for the novice programmer. It attempts to anticipate questions as well as explain the facts in order to provide a clear understanding of the language. The components of MiniPascal are taught in a logical, structured manner. Thus, advanced topics are explained only after the fundamental concepts have been provided.

The manual is divided into various units. The first four units explain the implementation of MiniPascal while the remaining units discuss subroutine information. Appendices are provided at the end as a reference tool for the reader.

Notation This manual uses typographic methods to distinguish between various symbols and words in MiniPascal.

- Ordinary English is printed in plain Roman letters, the kind that you are reading now.
- Special technical terms are printed in **boldface** when they are first defined. After that they are treated as ordinary English.
- Examples are printed in computer voice in order to distinguish them from ordinary text.
- Reserved words are printed in ALL CAPITALS.
- The names of data types are printed in *italics*.
- The symbol '¬' is used as a connector between two statement lines which should be written on one line. For example the text line:

RECT (0,1,¬
1,0);

should be read and entered in the computer as:

RECT (0,1,1,0);

Unit I Using MiniPascal as a File Format

Introduction

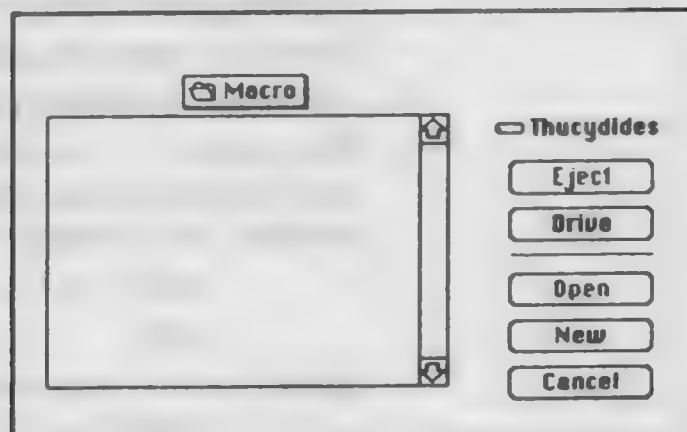
A MiniPascal File format is a text file (a file that stores characters) which contains only pre-defined MiniPascal language commands. These commands, when executed within MiniCad+, can be used to create graphic objects or to change graphic format settings (e.g. unit, scale and page settings). Using MiniPascal as a file format differs from using it as a programming language in that the file format method does not require any programming knowledge (i.e. no formal programming rules need to be followed). Therefore, the user is executing commands already written (pre-defined routines) as opposed to writing his own.

File formats may be created in two ways. First, the user may enter the pre-defined routines into a text file and then execute them from within MiniCad+. Secondly, he may convert a current MiniCad+ graphic display or file into a text file which represents it.

Section 1-1 Beginning Steps

MiniCad+ 3.0 assists the user in creating MiniPascal routines through the use of Command Editor in the program. Routines (Commands) should be placed into Command Palettes. All Procedures and Functions listed in this manual are available through the Command Editor. We recommend that users familiarize themselves with the Command Editor before trying to create code outside of the program.

Included with the program is the text editor, PEdit, which may be used to read and write MiniPascal code while working in this part of the manual.



When you select the **New** button, a blank text editing window will appear. Type the following line into the window:

```
RECT(0,1,1,0);
```

Now, select **Save** from the **PEdit** menu and name the file 'FirstFileFormat'. Exit the desk accessory by clicking in the close box located in the upper-left corner of the editing window.

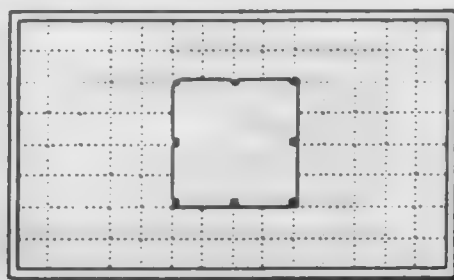
Select the **File** menu within **Minicad+**. Under the **File** menu you will find a menu item entitled **Import**. Move the mouse to the right until you see five more menu items: **PICT**, **PICT as Picture**, **Text Format**, **MiniCad 3D** and **DXF**. Select the **Text Format** menu item.



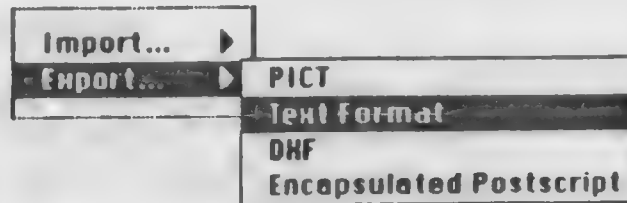
A file dialog box will appear and request that you select a text file. Select the text file named 'FirstFileFormat'. If the above statement was typed in correctly, a rectangle, as shown in Diagram 1-1, will appear on the screen. Otherwise, a dialog box will appear which will explain that "A MiniPascal error has occurred." If an error does occur, enter the **DA PEdit** and open the file called 'Error Output.' This file is a text file which will show you precisely what caused the error by displaying the incorrect statement line and the corresponding error description. Correct the error within the text file 'FirstFileFormat' and import the text file again in order to create the rectangle. You have just used **MiniPascal** as a file format. You executed a pre-defined routine (**RECT**) from a text file named 'FirstFileFormat'.

Diagram 1-1

FirstFileFormat Output



Now, select the File menu once again and choose the Export menu item.



Move the cursor to the right and choose the Text Format item. A dialog will appear which will ask you if you wish to save the created text file under the name 'MiniCad Startup.' Type the file name 'Text Output' and select the Save button. Enter PEdit and open the text file 'Text Output.' The text file will look as follows:

{Global Characteristics}

```
DrwSize(1,1);
Units(1,16);
GridLines(1/2");
PenGrid(1/16");
DoubLines(1/4");
SetOrigin(0,0);
Snap(TRUE,TRUE,TRUE);
TextFont(3);
TextSize(12);
TextFace([]);
TextSpace(2);
TextJust(1);
OpenPoly;
```

{End of Global Characteristics}

{Symbol Library Entries}

{End of Symbol Library Entries}

{Layer Characteristics}

```
Layer('Untitled');
SetScale(1.000000e+0);
ShowLayer;
CopyMode(8);
FillFore(0,0,0);
FillBack(65535,65535,65535);
```

```
PenFore(0,0,0);  
PenBack(65535,65535,65535);
```

```
{End of Layer Characteristics}
```

```
{Object Creation Code}
```

```
NameClass('None');  
PenSize(1);  
PenPat(2);  
FillPat(1);  
Rect(0',1",1",0');
```

```
{End of Creation Code}
```

The previous lines of code represent the MiniCad+ file which contained the rectangle that you created from the text file 'FirstFileFormat'. If the file 'Text Output' was imported within MiniCad+, a rectangle would appear which would be the same size and located in the same location as the original rectangle which was displayed on the screen.

You may wonder why the file 'Text Output' contains more routines than your original file 'FirstFileFormat', which contained only one. When a text file is created from a MiniCad+ file or display, all of the information relating to the file is written out. The first several routines store **global characteristics**. Global characteristics are file settings, such as the file's unit setting, its page size, origin and gridline width. The next routine section stores **symbol library entries**. However, this particular file has no entries. The file also stores **layer characteristics**, such as the name of the layer, its scale setting and its color. Finally, the file stores all **object characteristics**. Not only the object's location and size are stored, but also its color, fill pattern, line weight, etc. If a text file is imported into a MiniCad+ file which does not contain global or layer information, then the MiniCad+ file's current settings are used. The routines listed from the file 'Text Output' will be explained in detail in Unit V, 'MiniPascal Pre-defined Routines'.

Section 1-2 Getting Started

As we begin exploring MiniPascal, two terms need to be clarified. First, when the manual uses the term **programmer**, it is referring to you the reader (the person programming with MiniPascal). Secondly, when it specifies the term **user**, it is referring to the person executing the routines that you have written (the programmer and the user can obviously be the same person).

A **statement** is a programming language's unit of activity. Several statements act as a series of instructions for the computer to execute, or carry out. Type Statement Example 1-1 into the Command Dialog window and execute the commands as described in the section "Beginning Steps".

Statement Example 1-1

RECT(-1,1,0,0);	{Creates a rectangle}
OVAL(0,-0.5,1,-1.5);	{Creates an oval}

In the above example, two statements are listed. When executed, the first statement will create a rectangle with the upper-left corner at the coordinate (-1,1) and the bottom-right corner at the coordinate (0,0). The second statement will create an oval. The oval will be contained within a rectangular boundary with its upper-left corner at coordinate (0, -0.5) and its bottom-right corner at coordinate (1, -1.5). Please note that values inside of the parentheses require a digit—even if it is a zero—on the left side of a decimal point.

Important Note:

All values in MiniPascal require a digit—even if it is a zero—on the left side of a decimal point.

Each statement listed in the example is comprised of a **standard procedure**. A standard procedure is a built-in command which cannot be changed or modified by the programmer. When it is called (or used), it executes a particular series of statements for a specific purpose. For example, when the procedure RECT is called, it executes several statements that make the necessary calculations to create a rectangle with the dimensions provided in the parentheses. Please notice the semicolons which are lo-

cated at the end of each line in the example. The specifications of MiniPascal require that a semicolon be placed between any two statements or parts of a subprogram.

Important Note:

A semicolon must be placed between any two statements or parts of a subprogram.

**The Absolute,
Relative and
Distance-Angle
Method**

The location of each graphic object is determined by point (X,Y) designations provided by the programmer. As seen in the previous example, the procedures RECT and OVAL each required two point values (X1, Y1, X2, Y2) in order to determine their locations. Each graphic object's points may be specified in one of three ways: the absolute, relative or distance-angle method.

The absolute method of determining points is the method seen in the examples thus far. Each point (X,Y) is composed of two numeric values. These numeric values represent graphic coordinate locations (these coordinate values are displayed in the coordinate display boxes near the bottom of the MiniCad+ screen). Thus, if the user specifies the coordinate (0,0), the point given is where both the x and y coordinates equal zero.

The programmer specifies that he is using the absolute method by typing the word 'ABSOLUTE' prior to entering coordinate values. Therefore, if the user wanted to create a rectangle which has its top-left corner located at (1,2) and its bottom-right corner at (2,1), he would enter the following statements:

ABSOLUTE;	{Set the computer in absolute mode}
RECT(1,2,2,1);	{Create a rectangle}

Important Note:

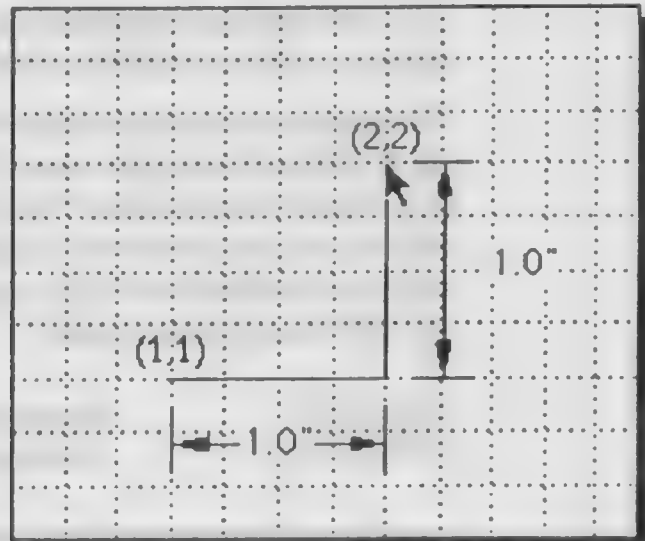
The absolute mode is the default mode. Therefore, if the programmer does not specify a mode, the absolute mode will be used.

Numeric values which are specified using the absolute method may be represented in different formats. These formats are shown in Table 1-1.

Table 1-1**Absolute Value Formats**

<i>Absolute Format</i>	<i>Example</i>
Integer	RECT(-1,1,1,-1)
Decimal	RECT(-1.5,0.5,0.5,-1.5)
Scientific Notation	RECT(15E-1,25E-1,35E-1,15E-1)
Fraction	RECT(-3/2,1/2,1/2,-3/2)
Engineering	RECT(0.5",0.5",1' 0.5",-3' 5")
Architectural	RECT(1/2",1/2",1' 1/2",-3' 1/2")
Metric	RECT(0.5m,0.5m,1.5m,-3.5m)

The **relative method** of specifying point designations looks similar to the absolute method. However, the relative method moves the graphics pen (the imaginary drawing tool) relative to its previous location. For example, if the graphics pen was located at the coordinate (1,1) and the programmer entered the point designation (1,1), the graphics pen would move one unit horizontally and one unit vertically to the absolute location (2,2).



The programmer specifies that he is using the relative method by typing the word 'RELATIVE' prior to entering point values.

```
MOVETO(1,1);           {Move the graphics pen to (1,1)}
RELATIVE;              {Set the computer in relative mode}
POLY(1,0,0,1,1,0,0,-2,-2,0); {Create a polygon}
```

If the programmer executes the above statements, the

graphics pen is initially located at the absolute coordinate (1,1). The computer is then told to execute all future point designations in relative mode. The graphics pen is moved one horizontal unit away from (1,1) in order to create the first point of the polygon (this location is (2,1)). The graphics pen is then moved one vertical unit in order to create the second point of the polygon (this location is (2,2)). Proceeding coordinates create additional polygon points which are placed relative to each previous point.

Numeric values which are written using the relative method may be represented in any of the absolute formats shown in Table 1-1.

The distance-angle method allows the user to provide a distance and an angle value for each point. Type Statement Example 1-2 into PEdit and execute it.

Statement Example 1-2

MOVETO(0,0);	{moves graphics pen to (0,0)}
RECT(2",#90,2",#0);	{creates a rectangle}

The first coordinate value of Procedure RECT is a distance value. In this example, the distance value equals two inches. This means that the top-left corner point of the rectangle will be two inches away from the coordinate (0,0). The next parameter is the angle value. Note that all angle values are preceded by a pound sign (#).

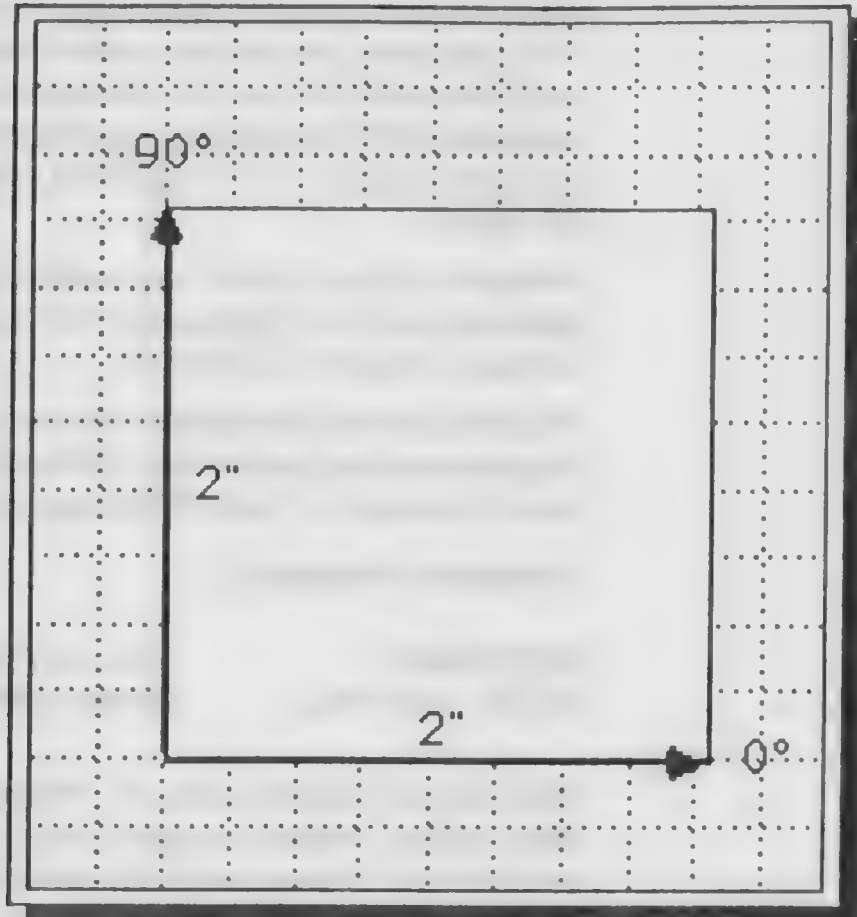
Important Note: All angle values are preceded by a pound sign (#).

This value determines which direction the line will be placed from the starting value. In this case, the line, which has a distance of two inches, will be at a 90 degree angle from the point location (0,0). The third and fourth parameters are distance and angle values for the bottom-right corner of the rectangle. Their location is also relative to the starting location point (0,0). The programmer may specify any starting location point through the use of the procedure MOVETO. Procedure MOVETO moves the graphics pen to the indicated coordinate location (this routine will be described in more detail in Unit V, 'MiniPascal Pre-defined Routines').

The output from Statement Example 1-2 is shown in Diagram 1-2.

Diagram 1-2

Rectangle Using the Distance-Angle Method



The distance and angle values used in the distance-angle method are also capable of having different formats. These formats are provided in Table 1-2.

Table 1-2

Distance-Angle Value Formats

<i>Distance Formats</i>	<i>Example</i>
Integer	RECT(2,#90,2,#0)
Decimal	RECT (2.0,#90,2.0,#0.0)
Scientific Notation	RECT (15E-1,#90,35E-1,#0)
Fraction	RECT (3/2,#90,1/2 ,#0)
Engineering	RECT (1'0.5",#90,0.5",#0)
Architectural	RECT (1'1/2",#90,1/2",#0)
Metric	RECT (1.5m,#90,0.5m,#0)

<i>Angle Formats</i>	<i>Example</i>
Integer degrees	RECT(2, #90, 2, #0)
Decimal degrees	RECT (2, #89.5, 2, #359.5)
Degree symbol	RECT (2, #90°, 2, #0°)
Degrees, Minutes, Seconds	RECT (2, #90d30'0", 2, #0d30'0")
Gradians	RECT (2, #100g, 2, #0g)
Radians	RECT (2, #1.57r, 2, #0r)
	RECT (2, #1/2 π , 2, #0 π)
Surveyor's units	RECT (2, #N 45d30'0" E, 4, #S- 45d30'0" E)

The examples provided in Table 1-2 are only a few compared to the many combinations possible. Distance and angle types can be used together in any fashion applicable (e.g. metric units could be used with an angle in surveyor's units, fractions could be used as a distance value with an angle's value in radians).

Surveyor's units deserve further attention. Surveyor's units appear in the following form:

<direction (N,S)> <angle value> <direction (E,W)>

The angle value is represented in a degrees/minutes/seconds form and the letter "d" is used to represent degrees.

When using an angle value between two bearings, such as #N 45d30'10" E, the angle value should not exceed 90 degrees. Therefore, 220 degrees equals #S 50d W. If the angle is precisely North, South, East or West, then simply use the single letter representing that direction. Also, 45° angle values between two bearings do not need to be written. Therefore, 45° equals #NE and 135° equals #NW. Execute Statement Example 1-3 to demonstrate this process. Statement Example 1-3 creates the same output as shown in Diagram 1-2.

Statement Example 1-3

RECT(2, #N, 2, #E); {creates a rectangle}

The best way to understand how MiniPascal works is to experiment with it. Create various objects using the

creation routines described in Unit V, 'Creation Routines'. Each procedure's purpose is explained and demonstrated. The following exercises are included in order to allow you to get experience using MiniPascal as a file format. The solutions to the exercises are located in the section entitled, aptly enough, 'Solutions to Exercises'.

Unit I Exercises

1. Use the RECT procedure to create three rectangles of equal size and align them horizontally.
2. Use the OVAL procedure to create a circle.
3. Use the RRECT procedure and the Distance-Angle Method to create a rounded rectangle.
4. Use the BEGINGROUP and ENDGROUP procedures to group an oval and a rectangle.
5. Use the POLY procedure to create an octagon.

Unit II Variables and Data Types in MiniPascal

Introduction The variables that MiniPascal uses are like the memory keys of a hand calculator—they store values. Variables in MiniPascal, however, have several advantages. First, unlike a calculator which has a fixed number of storage keys, we are able to create as many variables as we need. Secondly, we are able to individually name each variable. This ability allows us to use each name as a memory aid to clarify the purpose of each variable. Finally, we are not restricted to merely storing numbers. Variables can hold different types of data, including *integers* (whole numbers), characters (like letters, punctuation marks and single digits), *real* numbers (numbers with decimals or exponents), logical values (values which are either true or false, they are called *boolean* values), locations in memory (called *handles*), *arrays* (a structure which stores several elements) and *text* files (files which store characters). This unit describes the various types of variables. Use of variables within a subprogram is discussed in Section 3-3.

Section 2-1 Data Types

The various types of data are divided into several distinct groups: **Simple types**, which include *real* types, *string* types and *scalar* types (*integer*, *longint*, *boolean* and *char* types); **handle types**; and **structured types**, which include *arrays*, *vectors* and *text* file types. The various types are defined as follows:

Type	Description
integer:	integer variables store integers—the positive and negative counting numbers (-2,-1,0,1,2). The range of the type <i>integer</i> is the set of values from -32768 to +32767. An <i>integer</i> takes up two bytes of storage.
longint:	longint variables are able to store integers whose values exceed the range of the type <i>integer</i> . The range of the type <i>longint</i> is the set of values from -2147483648 to +2147483647. A <i>longint</i> takes up four bytes of storage.
boolean:	boolean (boo'-lee-an) variables, sometimes called logical variables, have one of two values—true or false. They are used with control structures which will be discussed in Section 3-3.

3-4. A *boolean* takes up one byte of storage.

- char:** char variables can hold any of the characters—letters, punctuation marks, or numerals—that appear on the keyboard. A *char* takes up two bytes of storage.
- real:** real variables store positive and negative numbers that include decimal points or are expressed as powers of 10 (-3.55,0.0,187E+2). The magnitude of real type values can range from approximately 1.9E-4951 to 1.1E4932 in scientific notation. A *real* type variable takes up ten bytes of storage.
- string:** a string value is a sequence of characters. It has a value in the range 1..255 characters. A *string* type variable takes up (the number of characters in the string + 1) bytes.
- handle:** a handle variable is used to access or change information within an existing graphic object in a document.
- array:** an array variable is used to store a group of values which can be accessed by location rather than by name. An array can store *integer*, *longint*, *real*, *string*, *char*, *boolean* or *handle* values.
- vector:** a vector is a MiniPascal data type which stores an x, y and z coordinate location. This data type allows programmers to perform standard mathematical vector operations within the language.
- text file:** a text file is MiniPascal's form of storing information. It contains char and string values.

One may raise the question as to why values are grouped into different types in MiniPascal. Type separation aids the computer in making useful error inspections. When a program is interpreted, an automatic type-checking mechanism looks for illogical activity. For example, if the compiler sees that the programmer is attempting to add an *integer* type to a *boolean* type '7 + true', then it will issue an error message. Having the interpreter "look over" our program provides us the opportunity to correct mistakes in the program before we run it.

Type-checking also occurs while the program is running. Therefore a fundamental restriction exists when using MiniPascal variables: Values must be of the same type as the variables that store them. This is especially true when writing your own subroutines because each variable that you decide to use must be declared. This means that prior to executing the statements of your subroutine, the computer must be told what variables you intend to use and their types. This process will be examined in more detail in Section 3-3.

Important Note: Values must be of the same type as the variables that store them.

Unit II Exercises ■ Name the variable type of each example below:

1. 100,000
2. FALSE
3. 'K'
4. 'England'
5. 1.45

Unit III Using MiniPascal as a Programming Language

Introduction By purchasing MiniCad+, you have obtained a remarkable graphics software tool. Imagine a graphics software program which allows each user to personalize it to fit his own needs. Imagine the ability to perform a particular configuration that *you* created by merely selecting a command in a palette.

MiniPascal is a programming language which allows each user to write his *own* subprograms which may be executed within the environment of MiniCad+. Its easy application and diversity of use will provide many beneficial options.

In order for a user to begin writing his own subprograms, however, he must be familiar with the material in the unit 'Using MiniPascal as a File Format'. This section provides an introduction to the proceeding programming information and defines many of the terms used.

Section 3-1 Symbols

As already mentioned, one advantage of the MiniPascal language in connection with MiniCad+ is that the user is capable of writing his own subroutines. These routines work in the same manner as the previously discussed standard procedures. During the next several units we will discuss the process of writing subroutines and examine the meaning and significance of each of their components.

Hopefully, you have experimented with using several creation procedures already. The non-programmer may question what relevance programming has compared to merely executing the pre-defined routines contained in the later units of this manual. The main advantage to programming is its versatility in problem solving. It is true that in using MiniPascal as a file format, the user can not only create various objects, such as rectangles, ovals, etc., but that he can also set the basic configurations of a MiniCad+ file, such as its page size, scale and unit settings. However, the pre-defined routines may not directly perform a particular action that you wish to accomplish. For instance, one may wish to design a routine which will automatically calculate and generate the proper locations of steps within a staircase. Thus, through programming, the user can write his own rou-

tines, solving his own distinct problems, being limited only by his imagination.

Let's start the programming section off with, what else, a program. Actually, it is not a program, but rather a subprogram. Since MiniCad+ is the main program, all programming done by the programmer will be in the form of subprograms (called procedures and functions). This distinction is made in order to clarify the programming characteristics of MiniPascal for those programmers who have knowledge of Pascal programming. Type Subprogram Example 3-1 into a command using the Command Editor. Run the command by double-clicking on it in the command palette. If an error occurs, view it using 'View Error' and make the corrections. Run the command again.

Subprogram Example 3-1

```
PROCEDURE MyFirst;  
BEGIN  
    WRITELN('This is my first MiniPascal subprogram.');
```

```
END;
```

```
Run(MyFirst);
```

Select the item and then use PEdit to open the text file entitled, 'Output File'. This file is the default output file for MiniCad+. It will contain the character string shown in Subprogram Example 3-1. For this example, we sent the information to the default output file. However, as will be explained in Unit IV, it is possible to designate a specific file to store output information.

Subprogram Example 3-1 is a short subprogram whose effect and output you probably figured out before you even executed it. We will begin learning MiniPascal by looking at what components comprise procedure MyFirst.

Important Note: The user may discontinue either the compiling process (importing a text file) or the execution process (selecting the macro menu) by depressing the command-period keys.

The MiniPascal language is composed of symbols. Symbols are the smallest meaningful units of source text in a subprogram. There are various types of symbols. Special symbols and reserved words are symbols having fixed meanings. If you try to change their meanings or use them in ways other than their intended uses, the interpreter (which interprets your commands into instructions the computer will understand) will issue an error. The following single characters are special symbols:

+ - * / = < > . ' , () : ; { } | & # ^ \$ • @ []

The following character pairs are also special symbols:

<> <= >= := .. **

Subprogram Example 3-1 used several special symbols, such as left and right parentheses and apostrophes. These special symbols and their uses will be explained in Sections 3-2 through 3-5.

Reserved words are the basic vocabulary of a language. They are also designated for a specific purpose and thus may not be redefined. Subprogram Example 3-1 used three reserved words: PROCEDURE, BEGIN and END. MiniPascal's reserved words are listed below. Their meaning and applications are discussed in later sections of this manual. These words will be capitalized throughout this manual. However, MiniPascal is not case sensitive—corresponding uppercase and lowercase letters are equivalent.

Important Note:

MiniPascal is not case sensitive.

Important Note:

In subprograms, such as in Subprogram Example 3-1, the name within the parentheses after the procedure Menu-Item must correspond to a previously created procedure. Also, note that the menu item name will retain its case characteristics when entered into the macro menu.

Table 3-1

Reserved words

AND	DO	FUNCTION	MOD
THEN	WHILE	BEGIN	ELSE
GOTO	NOT	TO	CONST
END	IF	PROCEDURE	UNTIL
DIV	FOR	LABEL	REPEAT
VAR	ARRAY	OF	

Important Note:

The user *may* redefine pre-defined subroutine names. For example, one could use the pre-defined name, 'RECT', to store a value. However, once a name is redefined, the user will lose the capability of using that particular routine. Therefore this action is not recommended.

Also, although pre-defined subroutine names may be used to store values, they may not be used as subprogram names. Therefore, the user may not call one of his subprograms RECT. A list of the pre-defined routines and other special words which should not be redefined are listed in the Appendices.

Identifier

Another type of symbol is the identifier. Identifiers are the names that denote variable types, procedures and other MiniPascal components which have yet to be discussed. For example, the name of Subprogram Example 3-1, MyFirst, is an identifier. Here are the rules for writing identifiers:

1. An identifier can be of any length, but only the first 10 characters are significant.
2. They are not case sensitive; corresponding upper case and lowercase letters are equivalent.
3. They may contain only letters, digits and under scores; in particular, they may not contain spaces.
4. Every identifier must begin with a letter.

The following examples of identifiers are legal in Mini-Pascal (they will not cause a compiler error):

A_Section CalculationWork RESULT2 get_recordvalue

Numbers Numbers are also symbols. Within a MiniPascal subrou-tine, you can use ordinary decimal notation for numbers that are constants of the data types *integer*, *longint* and *real*. You can also use scientific notation for *real* types. Here are various examples of correct notation for numbers in MiniPascal.

7 +780 -0.8 5E-7 49.21e+4 4e-3 3e2

Other symbols include labels. A label is a digit sequence in the range 0..9999. Leading zeros are not significant in labels. Labels are used with GOTO statements, which are described in Section 3-4.

Symbols also include **quoted string constants**. A quoted string constant (or **string**) is a sequence of zero or more characters from the ASCII character set. The following are the rules for writing strings:

1. Each must be enclosed by single quotes (apostrophes).
2. Blanks count as characters in quoted string constants.
3. The maximum number of characters in one constant is 255.
4. A quoted string constant with nothing between the single quotes denotes the null string.
5. If you want the quoted string constant to contain a single quote, you must write the single quote twice.

We saw a quoted string constant in Subprogram Example 3-1: 'This is my first MiniPascal subprogram.' These are other examples of quoted string constants:

'Australia' 'THUCYDIDES' 'Don't Worry' 'R' '' ' ' "

The last example is a null string and the next to last contains one single quote.

All string values have a length attribute. In the case of a quoted string constant, the length is fixed; it is equal to the actual number of characters in the string value. Therefore, the length attribute of the quoted string constant 'hello' is five. Another symbol, a **quoted character constant** is simply a quoted string constant whose length attribute is 1.

Delimiters Delimiters are symbols that separate other symbols in the source text so that the interpreter can distinguish them as discrete objects. Blanks (spaces, tabs and carriage returns) are the principal delimiters. In addition, all the special symbols listed earlier in this unit serve as delimiters while performing their other functions. Hence the interpreter can process the expression

`number_of_planets:=7+two;`

even though it contains no spaces or tabs, because + and := are delimiters.

Comments Comments also act as delimiters. The construct

`{ this section calculates the area }`

is called a comment. It is ignored by the interpreter. Comments allow the programmer to make remarks within his subprogram's code without affecting its execution. One important note is that a comment cannot be nested within another comment. Thus, the comment, {This is {nesting}} is not valid.

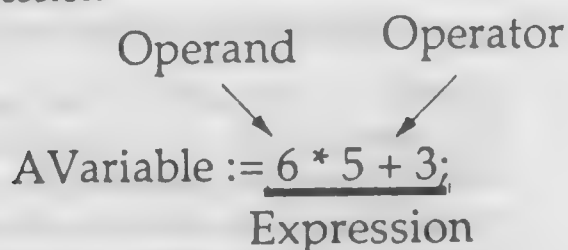
Important Note: A comment cannot be nested within another comment.

Section 3-2 Expressions

In MiniPascal, any representation of a value is an expression, whether it's a numeral, variable identifier or a sequence of numeric values and operation signs. An expression contains two components: **operators** and **operands**. Constants, variables and function calls (constants and function calls will be discussed shortly) are referred to as operands while the special symbols described previously (such as "+" and "**") are operators. Diagram 3-1 presents an expression.

Diagram 3-1

An Expression



Binary arithmetic operators are rules for combining operand values into new expressions. This is just a formal way of saying that these operators perform addition, subtraction, multiplication, exponentiation, and division. We need to distinguish between operators which may be used when the result from an expression is of type *integer*, *longint* or *real*. Type in Subprogram Example 3-2 and execute it as discussed in Unit I. Look at the results from the subprogram in the file "Output File."

Subprogram Example 3-2

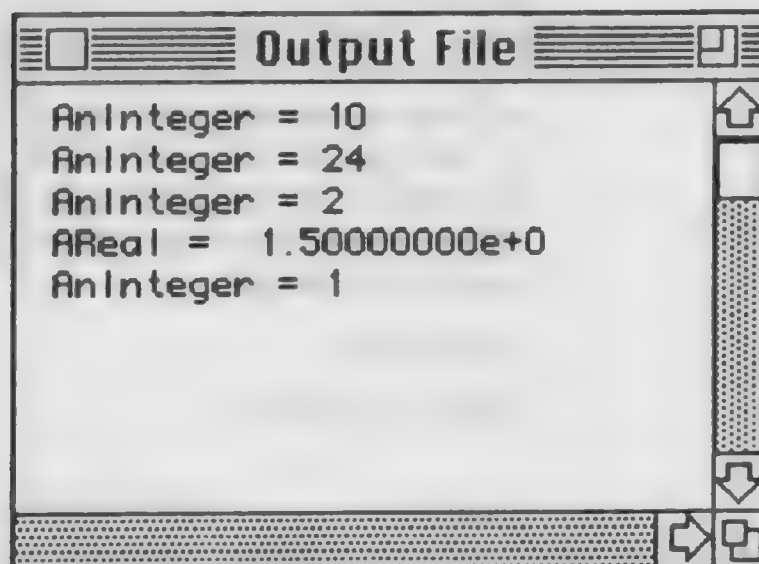
```
PROCEDURE MySecond;
VAR
  AReal : REAL;
  AnInteger : INTEGER;
BEGIN
  AnInteger := 6 + 4;
  WRITELN('AnInteger = ',AnInteger);
  AnInteger := 6 * 4;
  WRITELN('AnInteger = ',AnInteger);
  AnInteger := 6 - 4;
  WRITELN('AnInteger = ',AnInteger);
  AReal := 6/4;
  WRITELN('AReal = ',AReal);
  AnInteger := 6 DIV 4;
  WRITELN('AnInteger = ',AnInteger);
END;
Run(MySecond);
```

Important Note:

You may notice a symbol in the above example which seems strange. In MiniPascal, the assignment operator ($:=$) assigns a value to a variable. Therefore, in the first statement of Subprogram Example 3-2, the variable, *AnInteger*, stores the result of $6 + 4$. The assignment statement is the most frequently used statement in programming. It always takes the form:
variable identifier $:=$ the value represented by an expression;

The computer first performs a calculation by evaluating the expression on the right-hand side of the assignment. Then the result, or computed value, is stored to the variable on the left.

Output From Subprogram Example 3-2



From the output from Subprogram Example 3-2, we can see that we can take two values of type *integer* and add them, subtract them, or multiply them, and the result (or answer) will always be of type *integer*. Similarly we can take two values of type *longint* and find the same type of result. But, if we divide them, we can look at the result in two ways.

This is a *real* result

6 divided by 4 is 1.50000000e+00

This is an *integer* result

6 divided by 4 is 1 with a remainder of 2

Therefore, different operators are used with different types of values. The operators which can be used with *real* types are shown in Table 3-1.

Table 3-1
Real Operators

<i>Operator</i>	<i>Description</i>	<i>Example</i>
+	addition	7.4 + 8.7
-	subtraction	8.3 - 7.1
*	multiplication	7.0 * 8
** , ^	exponentiation	7.5**8
/	division	9 / 3

The integer operators are the same as the *real* operators except that the slash has been replaced by two special operators for integer division—DIV and MOD. The first, DIV, gives us the quotient of a division without any remainder. Such as: 6 DIV 4 is 1, 13 DIV 2 is 6. The second integer division operator, MOD, does just the opposite. It ignores the 'whole' part of the quotient, and provides only the remainder. Such as: 6 MOD 4 is 2, 13 MOD 2 is 1. Although DIV and MOD are reserved words in MiniPascal, they are thought of as symbols that represent operators, just like + and -. The *integer* operators are listed in Table 3-2.

TABLE 3-2
Integer Operators

<i>Operator</i>	<i>Description</i>	<i>Example</i>
+	addition	4 + 10
-	subtraction	17 - 6
*	multiplication	value * miles
** , ^	exponentiation	2**4
DIV	"whole number" division	9 DIV 4 (is 2)
MOD	"remainder" division	9 MOD 4 (is 1)

Longint operators correspond with the *integer* operators. All of the operators, their operations, operand types and result types are shown in Table 3-3:

Table 3-3
Operators

<i>Operator</i>	<i>Operation</i>	<i>Operand Type</i>	<i>Type of Result</i>
+	addition	<i>integer, longint, or real</i>	<i>same</i>
-	subtraction	<i>integer, longint, or real</i>	<i>same</i>
*	multiplication	<i>integer, longint, or real</i>	<i>same</i>
**	exponentiation	<i>integer, longint, or real</i>	<i>same</i>
/	division	<i>integer, longint, or real</i>	<i>real</i>
DIV	division with integer result	<i>integer or longint</i>	<i>same</i>
MOD	remainder	<i>integer or longint</i>	<i>same</i>

In MiniPascal, as in ordinary arithmetic, we can combine several small expressions in a chain of operations. For example, $2 + 3 - 4$ is a chain of small expressions. As long as the type rules are obeyed we can make expressions as long and complicated as we desire (arbitrary complexity). However, one question arises: what part of an expression is evaluated first? For example, look at the following expression:

$$20 * 15 - 5 \text{ DIV } 13 + 3$$

Completing the operations as they appear—20 times 15, minus 5, DIV 13, plus 3—gives 25 as the result. However, we could just as easily work from right to left and get the result -240. This situation brings us to the topic of **operator precedence** (which operators take place first). In MiniPascal, there is a pre-defined hierarchy (ordering) of precedence of operations. This hierarchy is listed in Table 3-4. In the above equation, $20 * 15$ will be the first part of the expression to be evaluated. This operation occurs for two reasons. First, because the multiplication operator has the highest order of precedence (other than DIV) in the expression. Secondly, although DIV has equal precedence, the multiplication operator is encountered first. Operators of the same precedence are executed from left to right. The single exception is for the exponentiation operator (** , $^{\wedge}$). Multiple exponentiation operations are performed right to left. The next set of operands to be evaluated in the above expression will be $5 \text{ DIV } 13$. Why, because the operator DIV has a higher precedence than the other operators in the expression. The final result of the above expression will be:

$$(300) - (0) + 3 = 303$$

Although there is a pre-defined order in which operators are executed within an expression, the programmer may define his own ordering through the use of left and right parentheses. Therefore, if the above expression was written:

$$20 * (15 - 5) \text{ DIV } (13 + 3),$$

the result would be:

$$20 * (10) \text{ DIV } (16) = 200 \text{ DIV } 16 = 12.$$

Table 3-4

Precedence of Operators

<i>Operators</i>	<i>Precedence</i>	<i>Category</i>
NOT, ** or ^	highest	exponent and unary operators (signs)
*, /, DIV, MOD, AND, &	second	"multiplying" operators
+, -, OR,	third	"adding" operators and signs
=, <>, <, >, <=, >=	lowest	relational operators

Variables of type *boolean* were previously discussed to be TRUE or FALSE. Five operators exist in MiniPascal for *boolean* type operations. These include: AND, OR, NOT, |, and &. We use *boolean* operators to create *boolean*-valued expressions. Type Subprogram Example 3-3 into a command file and execute it.

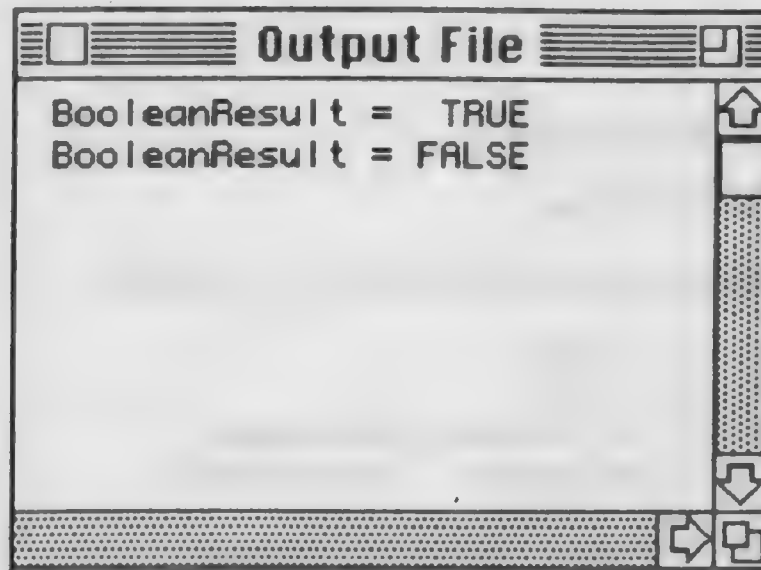
Subprogram Example 3-3

```

PROCEDURE MyThird;
VAR
  ABoolean1, ABoolean2, BooleanResult : BOOLEAN;
BEGIN
  ABoolean1 := TRUE;
  ABoolean2 := TRUE;
  BooleanResult := ABoolean1 AND ABoolean2;
  WRITELN(' BooleanResult = ', BooleanResult);
  ABoolean2 := FALSE;
  BooleanResult := ABoolean1 AND ABoolean2;
  WRITELN(' BooleanResult = ', BooleanResult);
END;
RUN(MyThird);

```

Output from Subprogram Example 3-3



If we have two *boolean* values (such as ABoolean1 and ABoolean2) the expression ABoolean1 AND ABoolean2 is evaluated as TRUE if both ABoolean1 and ABoolean2 are TRUE. If either or both of the values are FALSE, the entire expression is FALSE. These operations are shown in Table 3-5.

Table 3-5

Boolean Operator "AND" Results

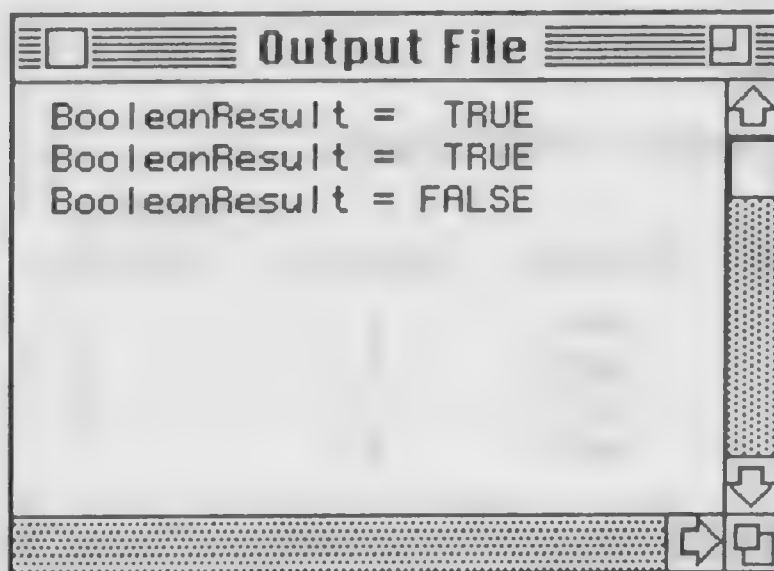
<i>ABoolean1</i>	<i>Operator</i>	<i>ABoolean2</i>	<i>Result of Expression</i>
T	AND	T	T
F	AND	T	F
T	AND	F	F
F	AND	F	F

Subprogram Example 3-4 demonstrates the use of the *boolean* operator OR. Enter Subprogram Example 3-4 into a command and execute it.

Subprogram Example 3-4

```
PROCEDURE MyFourth;  
VAR  
    ABoolean1,ABoolean2,BooleanResult : BOOLEAN;  
BEGIN  
    ABoolean1 := TRUE;  
    ABoolean2 := TRUE;  
    BooleanResult := ABoolean1 OR ABoolean2;  
    WRITELN('BooleanResult = ', BooleanResult );  
    ABoolean2 := FALSE;  
    BooleanResult := ABoolean1 OR ABoolean2;  
    WRITELN('BooleanResult = ', BooleanResult );  
    ABoolean1 := FALSE;  
    BooleanResult := ABoolean1 OR ABoolean2;  
    WRITELN('BooleanResult = ',BooleanResult);  
END;  
RUN(MyFourth);
```

Output from Subprogram Example 3-4



The boolean operator OR is less restrictive than AND. The expression ABoolean1 OR ABoolean2 is TRUE if either ABoolean1 or ABoolean2, or both of them are TRUE. It is only FALSE if both ABoolean1 and ABoolean2 are FALSE. See Table 3-6.

Table 3-6

Boolean Operator "OR" Results

<i>ABoolean1</i>	<i>Operator</i>	<i>ABoolean2</i>	<i>Result of Expression</i>
T	OR	T	T
F	OR	T	T
T	OR	F	T
F	OR	F	F

The ampersand(&) and vertical bar(|) are referred to as **short-circuit operators**. They are used in boolean expressions when it is desired to stop evaluating an expression as soon as its definite result is determined. For example, look at the statement below:

BooleanResult := ABoolean1 OR ABoolean2

Assume that the value of ABoolean1 is TRUE. The expression will be evaluated as TRUE regardless of the value of ABoolean2 (see Table 3-6). Thus, ABoolean2 does not need to be evaluated because its result will have no affect upon the result of the expression. However, using the OR operator in the above example, ABoolean2 will always be evaluated. If the above example is written:

BooleanResult := ABoolean1 | ABoolean2,

evaluation of the expression stops as soon as ABoolean1's TRUE value is reached. Likewise, in the expression:

BooleanResult := ABoolean1 AND ABoolean2,

if ABoolean1 is FALSE, the entire expression will be FALSE regardless of ABoolean2's value. The expression.

BooleanResult := ABoolean1 & ABoolean2

prevents ABoolean2 from being evaluated unnecessarily.

The result of the NOT operator is the opposite of its operand. For example,

BooleanResult := NOT ABoolean1

represents TRUE if ABoolean1 is FALSE and FALSE if ABoolean1 is TRUE. These results are shown in Table 3-7.

Table 3-7

Boolean Operator "NOT" Results

<i>Operator</i>	<i>Condition1</i>	<i>Result of Expression</i>
NOT	T	F
NOT	F	T

The types of operands and results for Boolean operations are shown in the next table.

Table 3-8

Boolean Operators

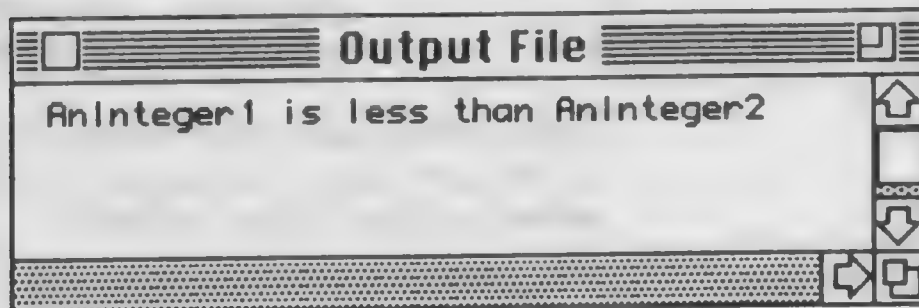
<i>Operators</i>	<i>Operation</i>	<i>Operand</i>	<i>Type of Result</i>
OR,	disjunction	boolean	boolean
AND, &	conjunction	boolean	boolean
NOT	negation	boolean	boolean

MiniPascal uses the fact that equalities and inequalities always make an assertion that has to be either TRUE or FALSE. This makes them boolean-valued expressions. For example, the expression $6 < 15$ is either TRUE or FALSE. Type in Subprogram Example 3-5.

Subprogram Example 3-5

```
PROCEDURE MyFifth;  
VAR  
    AnInteger1, AnInteger2 : INTEGER;  
BEGIN  
    AnInteger1 := 5;  
    AnInteger2 := 10;  
    IF AnInteger1 < AnInteger2  
    THEN  
        WRITELN('AnInteger1 is less than AnInteger2')  
    ELSE  
        WRITELN('AnInteger1 is greater than AnInteger2');  
    END;  
    RUN(MyFifth);
```

Output from Subprogram Example 3-5



Subprogram Example 3-5 and previous subprogram examples contain components which have yet to be discussed. This has not occurred unintentionally. All components of writing subprograms will be explained in the proceeding sections and units. The intention of the subprogram examples at this point is to merely give the reader experience in working with MiniPascal and to provide concrete examples of how the provided information can be used. Therefore, in Subprogram Example 3-5, the reader should not be concerned with precisely how IF..THEN..ELSE statements work, but rather that he is capable of comparing variables (IF AnInteger1 < AnInteger2) and that the result of this comparison is either TRUE or FALSE. Table 3-9 presents the types of operands and results for relational operators.

Table 3-9
Relational Operators

<i>Operator</i>	<i>Operation</i>	<i>Operand</i>	<i>Type of Result</i>
=	equal to	compatible simple types	boolean
<>	not equal to	compatible simple types	boolean
<	less than	compatible simple types	boolean
>	greater than	compatible simple types	boolean
<=	less than or equal to	compatible simple types	boolean
>=	greater than or equal to	compatible simple types	boolean

When the operands are numeric, they do not need to be of the same type if one operand is of type *real* and the other is of type *integer* or *longint*.

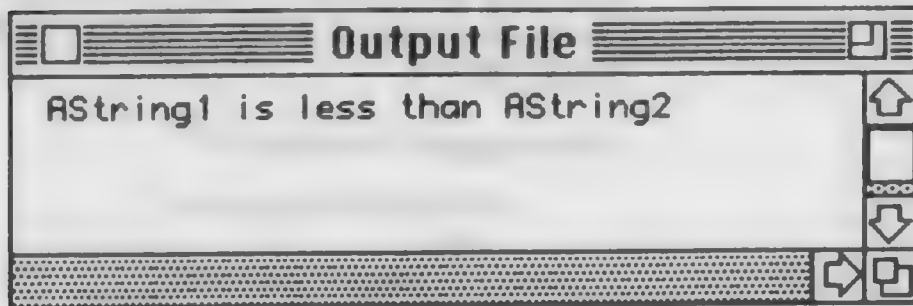
The operators provided in Table 3-9 may be used with not only numeric types, but also with booleans and strings. When comparing booleans, the following rules apply. If *ABoolean1* and *ABoolean2* are boolean operands, then *ABoolean1* = *ABoolean2* denotes their equivalence, *ABoolean1* <> *ABoolean2* denotes the logical exclusive-or operation, and *ABoolean1* < *ABoolean2* denotes the logical expression "ABoolean1 implies ABoolean2" (because FALSE < TRUE). You can also write NOT *ABoolean1* OR *ABoolean2* for logical implication.

Finally, when using the relational operators to compare strings, they are ordered according to the ordering of the ASCII character set. Type Subprogram Example 3-6 into a text file and execute it.

Subprogram Example 3-6

```
PROCEDURE MySixth;  
VAR  
    AString1,AString2 : STRING;  
BEGIN  
    AString1 := 'ABC';  
    AString2 := 'DEF';  
    IF AString1 < AString2  
    THEN  
        WRITELN('AString1 is less than AString2')  
    ELSE  
        WRITELN('AString1 is greater than AString2');  
END;  
RUN(MySixth);
```

Output from Subprogram Example 3-6



Note that any two string values can be compared because all string values are compatible. String comparisons follow these steps:

1. The two strings are compared a character at a time, starting with the first character.
2. Two corresponding characters are compared. If the ASCII value of one character is greater than the other, then the corresponding string is greater than the other. (The ASCII values of characters are presented in Appendix B, "Reserved Words and the ASCII Character Set").
3. If the two corresponding characters are equal, the point of comparison advances to the next character in each string and the process returns to step 2.
4. If the end of one string has been reached, its value is less than the other string.
5. If the ends of both strings have been reached, the two strings are equal.

Thus far, we have looked at operands which have been either numeric values or variables, but other types of operands exist as well. In Unit I, we made reference to standard procedures (built-in commands). However, standard functions also exist. A standard function may be thought of in the following manner. Usually on a calculator, there will be keys which square or find the square root of a value. These keys are called function keys. They make working with numbers much easier because a single keystroke completes an otherwise lengthy series of calculations. Well, MiniPascal provides the programmer with the capability of using either pre-defined or writing his own "function keys." These functions, contrary to calculator functions, can be used with not only numerical values but other types as well (such as *string* and *boolean* types). Standard functions are used through function calls. Take a look at Subprogram Example 3-7.

Subprogram Example 3-7

```
PROCEDURE MySeventh;
VAR
  AnInteger : INTEGER;
BEGIN
  AnInteger := 49;
  AnInteger := SQRT(AnInteger) + 4;
  Writeln('AnInteger = ',AnInteger);
END;
RUN(MySeventh);
```

In the Procedure MySeventh, we see that the statement part, `SQRT(AnInteger)`, is a function call. A function call usually has two parts—the name of the function, followed by the function's parameter(s) in parentheses. For example, in the above example, the function's identifier (name) is `SQRT` and its parameter is `AnInteger`. The entire function call represents the value of 7. Therefore, a function call is an operand, just as a numeric value is. The statement

`AnInteger := SQRT(AnInteger) + 4;`

will be evaluated the same way as if the statement was written

`AnInteger := 7 + 4.`

Always remember that a function call is an operand, regardless of its name or the number of parameters given with it. Function calls and function declarations will be discussed in detail in Section 3-5. (The pre-defined functions available in MiniPascal are listed and described in Unit V).

Important Note: A function call is an operand.

Section 3-3 Variables and Constants

Up to this point we have made several references to variables and we have used them in many of the previous subprogram examples. We mentioned that variables store values and that they must be declared in a subroutine so that the computer will know what type the variable is. In this section we will explain how to declare variables within a subroutine and we will also introduce a different form of storing values called constants.

A variable declaration provides a variable's identifier and type. The declaration begins with the reserved word **VAR**, which is a shorthand for variable. Then, each identifier is listed along with its type—the kind of value the variable will hold. A colon separates the declared variable and its type and a semicolon is located after each declaration. See the example below:

```
VAR
  AnInteger      : INTEGER;
  AChar          : CHAR;
  AReal          : Real;
```

In order to declare two variables that are the same type, you may list the variables on the same line separated by a comma. For example, if you wish to declare the variables **AReal1** and **AReal2**, which are both of type *real*, you may declare them in the following manner.

```
VAR
  AReal1, AReal2 : REAL;
```

Where does the variable declaration belong? The first part of a subroutine is the heading—it contains the reserved word **PROCEDURE** and a procedure identifier (name). Next comes the declaration part, which contains the information that we just discussed. Finally comes the statement part, where the subprogram's action takes place. Referring back to Subprogram Example 3-7, a subprogram may be broken down into the following components:

PROCEDURE MySeventh;	{The Heading}
VAR	{The Declaration— Part}
AnInteger : INTEGER;	{Variable Declara— tions}
BEGIN	{The Statement — Part}
AnInteger := 49;	{A Statement}
AnInteger := SQRT(AnInteger) + 4;	{A Statement}
WRITELN('AnInteger = ',AnInteger);	{A Statement}
END;	

The values processed by MiniPascal may also be stored as constants. A constant differs from a variable in that once it has been defined, its value cannot be changed during the course of the subroutine. Constants must be defined as opposed to declared. A constant definition part begins with the reserved word CONST, followed by each constant's identifier, an equals sign, and then its value. A semicolon separates successive definitions. For example:

```

CONST
    AConstant1      =  4.2387;
    AConstant2      =  TRUE;

```

Note that an equals sign (=) is used in the constant declaration, and not the assignment operator (:=). The assignment operator is only used within the statement part of a subroutine.

With the introduction of constants, the earlier representation of the components of a subroutine was incomplete. Constants are defined in a definition part, prior to the declaration part. Constants are demonstrated in Subprogram Example 3-8:

Subprogram Example 3-8

PROCEDURE MyEighth;	{The Heading}
CONST	{The Definition Part}
AConstant = 4.298765875;	{Constant Definitions}
VAR	{The Declaration Part}
AReal : REAL;	{Variable Declara— tions}
BEGIN	{The Statement Part}
AReal := AConstant * 3 / 2;	{A Statement}
WRITELN('AReal = ', AReal);	{A Statement}
END;	
RUN(MyEighth);	

Why are constants used? When writing a subprogram, there may be some values which you know will remain the same throughout its execution. Then, you may ask, why don't we just use the actual numeric value in our subprogram as opposed to using a constant. The reason is that as programmers, we attempt to make our subprograms as adaptable and versatile as possible. If, in the future, this "numeric constant" changes its value, we merely need to change the constant in the declaration part as opposed to changing every numeric value in our statement part. Also, as seen in the above example, once a constant has been defined, a constant identifier can be used in place of the value it represents. This is particularly convenient when the constant is a long mathematical or physical value that is used many times during the course of a subroutine. Using a constant as opposed to a numeric value can save a programmer a great deal of time and energy.

Section 3-4 Statements

We briefly discussed statements in the "Getting Started" section of this manual. Statements are made up of expressions combined with certain reserved words. Statements perform the actual work of a MiniPascal subroutine, doing such things as giving a value to a variable or providing conditional execution of other statements. This section will discuss the various types of statements and their implementation.

As stated earlier, the assignment statement sets the value of a variable. The symbol `:=` can be read as "assigned to." When using assignment statements, several rules exist:

Rule #1: A *real* variable may be set to the value of another *real*, an *integer*, a *longint* or an expression yielding *integer*, *longint* or *real* results. Subprogram Example 3-9 presents various assignment statements applicable with a variable of type *real*.

Subprogram Example 3-9

```
PROCEDURE MyNinth;  
VAR  
  AReal1, AReal2 : REAL;  
BEGIN  
  AReal1 := 4.27;           {Assignment to a real value}  
  AReal2 := AReal1;         {Assignment to a real variable}  
  AReal2 := 40000;          {Assignment to a longint value}  
  AReal2 := 3;              {Assignment to an integer -  
                             value}  
  AReal2 := AReal1 * 4.3 + 2; {Assignment to a real expres-  
                             sion}  
END;  
RUN(MyNinth);
```

Rule #2: A *longint* variable may be set to the value of a *longint*, an *integer* or an expression yielding *longint* or *integer* results. Subprogram Example 3-10 demonstrates several assignment statements using a variable of the *longint* type.

Subprogram Example 3-10

```
PROCEDURE MyTenth;  
VAR  
  ALongint1, ALongint2 : LONGINT;  
BEGIN  
  ALongint1 := 35000;        {Assignment to a -  
                             longint value}  
  ALongint2 := ALongint1;    {Assignment to a -  
                             longint variable}  
  ALongint2 := 3;           {Assignment to an -  
                             integer value}  
  ALongint2 := 40000 + ALongint1; {Assignment to a -  
                             longint expr.}  
END;  
RUN(MyTenth);
```

Rule #3: A *boolean* variable may be set to the value of another *boolean* or to an expression yielding a *boolean* result. Several *boolean* assignment statements are shown in Subprogram Example 3-11.

Subprogram Example 3-11

```
PROCEDURE MyEleventh;  
VAR  
  ABoolean1,ABoolean2 : BOOLEAN;  
  AnInteger : INTEGER;  
BEGIN  
  AnInteger := 6;  
  ABoolean1 := TRUE;           {Assignment to a  
                                boolean value}  
  ABoolean2 := ABoolean1;      {Assignment to a  
                                boolean variable}  
  ABoolean2 := (AnInteger < 10); {Assignment to a  
                                boolean expr.}  
RUN(MyEleventh);
```

Important Note: In order to assign a boolean expression to a boolean variable, the boolean expression must be enclosed within parentheses.

Rule #4: A *char* variable may be set to the value of a *char* or another *char* variable. Subprogram Example 3-12 demonstrates several valid *char* assignment statements.

Subprogram Example 3-12

```
PROCEDURE MyTwelvth;  
VAR  
  AChar1,AChar2 : CHAR;  
BEGIN  
  AChar1 := 'A';           {Assignment to a char value}  
  AChar2 := AChar1;        {Assignment to a char variable}  
END;  
RUN(MyTwelvth);
```

Rule #5: A *string* variable may be set to the value of a *string* or another *string* variable. It may also be set to the value of a variable of type *char*. Subprogram Example 3-13 presents various assignment statements applicable with a variable of type *string*.

Subprogram Example 3-13

```
PROCEDURE MyThirteenth;  
VAR  
  AString1,AString2 : STRING;  
  AChar : CHAR;  
BEGIN  
  AChar := 'A';  
  AString1 := 'This is a string'; {Assignment to a string value}  
  AString2 := AString1;          {Assignment to a string vari-  
                                able}  
  AString2 := AChar;             {Assignment to a char vari-  
                                able}  
END;  
RUN(MyThirteenth);
```

Another type of statement which we have seen in the previous subprogram examples is the **procedure statement**. The procedure statement is a statement that is used to call a procedure. A procedure statement consists of a procedure identifier and its parameters (if it has any). Subprogram Example 3-14 demonstrates this statement type.

Subprogram Example 3-14

```
PROCEDURE MyFourteenth;  
BEGIN  
  OVAL(0,1,1,0); {Procedure statement with four -  
                 parameters}  
END;  
RUN(MyFourteenth);
```

You will recall that variables that are of type *boolean* have one of two values, TRUE or FALSE. Initially it may have appeared that this type did not seem very useful. However, boolean values may be the most useful values of all because they let subprograms incorporate decisions. These decisions are determined through the use of **conditional statements**. Let's take a look at Subprogram Example 3-15.

Subprogram Example 3-15

```
PROCEDURE MyFifteenth;  
VAR  
    AnInteger1 : INTEGER;  
BEGIN  
    AnInteger1 := 5;  
    IF AnInteger1 <= 10  
    THEN  
        Writeln('AnInteger1 is less or equal to 10')  
    ELSE  
        Writeln('AnInteger1 is greater than 10');  
    END;  
    RUN(MyFifteenth);
```

The statement part of Subprogram Example 3-15 may be viewed in the following manner:

```
IF AnInteger1 is less than or equal to 10  
THEN  
    action  
ELSE  
    alternative action
```

The IF structure lets a subprogram choose between taking two alternatives. When an IF structure is entered, its boolean expression is evaluated. If it's true, the THEN action is carried out, and the alternative ELSE action is skipped. If the boolean expression represents the value false, then the THEN action is jumped over and the ELSE action is executed instead. Execute Subprogram Example 3-15 and look at the results from the output. Then, change the boolean expression (`AnInteger1 <= 10`) to be false and view the results again. Notice that semicolons are not placed after statements immediately preceding the reserved word ELSE.

Important Note: Semicolons should not be placed after statements which immediately precede the reserved word ELSE.

IF..THEN..ELSE statements can be written to make complex decisions. Several boolean expressions can be evaluated by using the boolean operators discussed in Section 3-2. Subprogram Example 3-16 demonstrates the use of boolean operators with boolean expressions.

Subprogram Example 3-16

```
PROCEDURE MySixteenth;  
VAR  
  AnInteger1 : INTEGER;  
BEGIN  
  AnInteger1 := 8;  
  IF (AnInteger1 <= 10) AND (AnInteger1 >= 5) AND  
    (AnInteger1 <> 9)  
  THEN  
    WRITELN('The boolean expression is true')  
  ELSE  
    WRITELN('The boolean expression is false');  
END;  
RUN(MySixteenth);
```

Important Note:

When evaluating several boolean expressions with boolean operators, each boolean expression must be enclosed within parentheses.

From the above example, you can see that the boolean expression will only be true if AnInteger1 equals 5, 6, 7, 8 or 10. If this is the case (as it is in the above example), the statement

WRITELN ('The boolean expression is true')

will be executed. Otherwise, the statement

WRITELN ('The boolean expression is false');

will be executed. Subprogram Example 3-16 will execute only one statement dependent upon the result of the boolean expression.

However, what if the programmer wishes to execute several statements dependent upon the result of a boolean expression as opposed to just one? When writing MiniPascal subroutines, it is possible to treat several statements as if they were one by using a compound statement.

In order to explain compound statements, an explanation of the words BEGIN and END is necessary. BEGIN and END are reserved words which are required of any statement section (the executable part of a subroutine). The section always starts with BEGIN and terminates with END. When these reserved words are used with a sequence of statements, they create a single compound statement. Let's look at Subprogram Example 3-17.

Subprogram Example 3-17

```
PROCEDURE MySeventeenth;  
VAR  
    AnInteger1, AnInteger2 : INTEGER;  
BEGIN  
    AnInteger1 := 8;  
    IF (AnInteger1 <= 10) AND (AnInteger1 >= 5) AND  
        (AnInteger1 < 9)  
    THEN  
        BEGIN  
            WRITELN('The boolean expression is true');  
            AnInteger2 := 1;  
            WRITELN('AnInteger2 = ', AnInteger2);  
        END  
    ELSE  
        BEGIN  
            WRITELN('The boolean expression is false');  
            AnInteger2 := 0;  
            WRITELN('AnInteger2 = ', AnInteger2);  
        END  
    END;  
END;  
RUN(MySeventeenth);
```

Subprogram Example 3-17 is similar to Subprogram Example 3-16 except that depending upon the result of the boolean expression, several statements are executed as opposed to one. With regards to the THEN and ELSE actions, either may be composed of a single or compound statement. If a compound statement is used, then the BEGIN and END reserved words must be incorporated.

Important Note:

If a compound statement is used, then the BEGIN and END reserved words must be incorporated.

Although the ELSE option is mentioned above, it is not required with an IF ... THEN statement. If the boolean expression of an IF statement is false, then the computer will merely jump over the THEN statement(s) and continue executing the following statements. Therefore, Subprogram Example 3-17 could have been written as follows:

```

PROCEDURE MySeventeenth;
VAR
  AnInteger1,AnInteger2 : INTEGER;
BEGIN
  AnInteger1 := 8;
  IF (AnInteger1 <= 10) AND (AnInteger1 >= 5) AND~
    (AnInteger1 < 9)
  THEN
    BEGIN
      WRITELN('The boolean expression is true');
      AnInteger2 := 1;
      WRITELN('AnInteger2 = ',AnInteger2);
    END;
    WRITELN('After the boolean expression');
  END;
  RUN(MySeventeenth);

```

In an IF statement, the statement following the THEN word or the ELSE word can also be an IF statement and can contain its own THEN . . . ELSE clause. Thus an IF statement can be written to take different actions for each of several mutually exclusive conditions. The process of putting IF statements within other IF statements is called nesting. For example:

Subprogram Example 3-18

```

PROCEDURE MyEighteenth;
VAR
  AnInteger1,AnInteger2 : INTEGER;
BEGIN
  AnInteger1 := 108;
  IF AnInteger1 < 100
  THEN
    BEGIN
      IF AnInteger1 > 7
      THEN
        IF AnInteger1 < 9
        THEN
          WRITELN('AnInteger1 = 8');
        END
      ELSE
        BEGIN
          IF AnInteger1 < 200
          THEN
            BEGIN
              WRITELN('AnInteger1 is less than 200');
              IF AnInteger1 > 107
              THEN
                BEGIN
                  WRITELN('AnInteger1 is greater~
                    than 107');
                  IF AnInteger1 < 109

```

```

                                THEN
                                WRITELN('AnInteger1 = 108');
                                END;
                                END;
                                END;
                                END;
                                RUN(MyEighteenth);

```

Subprogram 3-18 demonstrates nesting by evaluating several boolean expressions to determine the value of the variable `AnInteger1`. As you can see, using boolean expressions within a subprogram allows the subprogram to decide which particular action to execute.

In the course of writing MiniPascal subroutines, the programmer may find it necessary to repeatedly execute a statement section. This process is sometimes referred to as "looping." MiniPascal is capable of setting up a loop and executing particular routines through the use of **repetition statements**. However, the user must determine the constraints on this process.

Important Note:

When using repetition statements, the programmer may enter what is called an "infinite loop." This event occurs when a repetition statement never comes to an end. If this does occur, depress the command and period key and this will stop the execution of the subprogram.

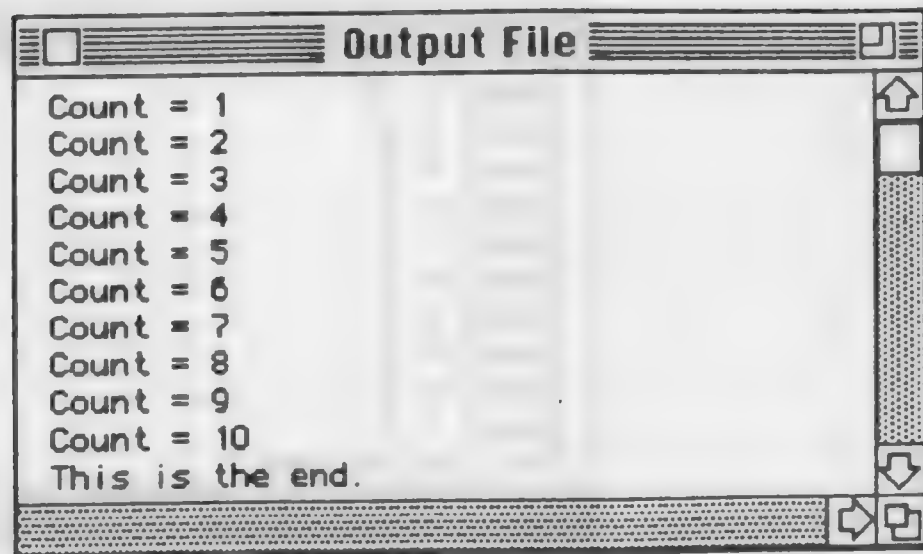
Subprogram Example 3-19

```

PROCEDURE MyNineteenth;
VAR
    Count : INTEGER;
BEGIN
    FOR Count := 1 TO 10 DO
        WRITELN('Count = ',Count);
        WRITELN('This is the end. ');
    END;
    RUN(MyNineteenth);

```

Output from Subprogram Example 3-19



```
Count = 1
Count = 2
Count = 3
Count = 4
Count = 5
Count = 6
Count = 7
Count = 8
Count = 9
Count = 10
This is the end.
```

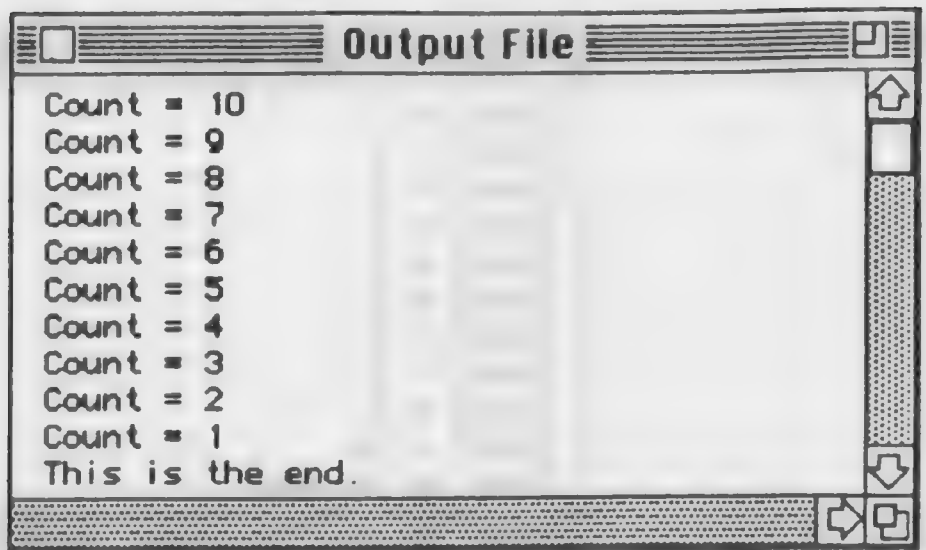
Subprogram Example 3-19 demonstrates the FOR looping structure. Execute the subprogram and view the output. When a FOR structure is entered, its counter variable (in this example, *Count*) is assigned to the initial counter value (in the above example, the value 1). As the counter variable is increased to the next *integer* value, the FOR structure's action (the first *writeln* statement) is carried out. When the counter variable represents the final value (10), the loop iterates (repeats) one last time, and the subprogram moves on to the next statement (the second *writeln* statement). The numeric values 1 and 10 in the FOR statement are called **limit expressions**. They specifically designate the initial starting and ending value of the counter.

Using the word **DOWNTO** in place of **TO** reverses the counting process. For this to work, however, the initial value must be greater than the final value. Execute Subprogram Example 3-20.

Subprogram Example 3-20

```
PROCEDURE MyTwentieth;
VAR
    Count : INTEGER;
BEGIN
    FOR Count := 10 DOWNTO 1 DO
        WRITELN('Count = ',Count);
        WRITELN('This is the end. ');
    END;
RUN(MyTwentieth);
```


Output from Subprogram Example 3-20



```
Count = 10
Count = 9
Count = 8
Count = 7
Count = 6
Count = 5
Count = 4
Count = 3
Count = 2
Count = 1
This is the end.
```

The initial and final value expressions, as well as the counter variable, must be of type *integer*. The limit expressions may be simple constants or variables, or complex expressions containing operators and functions. Also, as discussed before with the IF structure, several statements can be executed by the FOR structure through the use of a compound statement.

Some important rules exist with the FOR statement:

- Rule # 1. The counter variable must be a variable of type *integer* and it must be declared within the procedure it is used.
- Rule # 2. Do not try to change the value of the counter variable from within the FOR statement, doing so can have unpredictable results.
- Rule # 3. Do not include the counter variable in either of the limit expressions.
- Rule # 4. After the FOR statement is finished, the value of the counter variable may be unspecified. The counter variable should be reinitialized before being used in an expression.
- Rule # 5. The limit expressions are evaluated just once, before the first pass. Changing them from within the FOR statement will not alter its behavior.
- Rule # 6. If the limit expressions have equal value, the FOR statement will execute its control statement once.

Rule # 7. If the limit values are reversed—large limit less than small limit—the FOR statement will be skipped.

Like the IF control structure, the REPEAT structure uses a boolean expression to control the execution of an action. The REPEAT structure's action takes place, then its exit condition is evaluated. The loop's action is repeated until the exit condition is met.

If the expression that represents the exit condition is false, the exit condition is not met, and the loop's action is iterated (repeated). If the expression is true, the exit condition is met, the loop is terminated, and the program moves on to the next statement. In outline form the REPEAT structure looks like this:

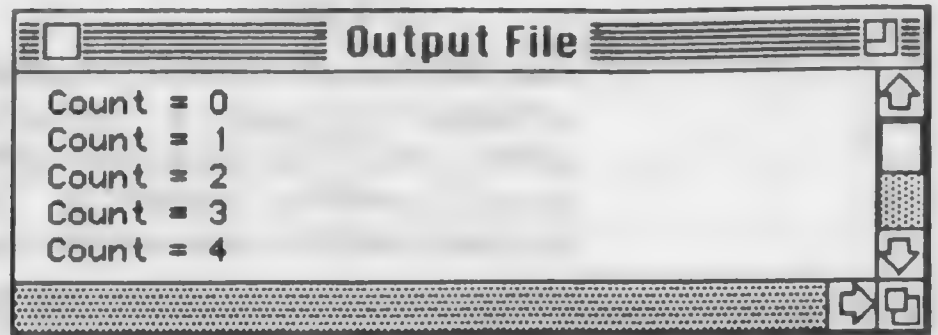
```
REPEAT
    action
UNTIL boolean expression ; [the exit condition]
```

One important note is that the action in a loop is an unbreakable unit. We will never exit from the middle of a loop. The exit condition is evaluated only after the loop action has been completely carried out. Also, REPEAT and UNTIL create their own compound out of the statements that they control so you do not need to use BEGIN and END. Subprogram Example 3-21 demonstrates the REPEAT..UNTIL looping structure. The statements between the REPEAT and UNTIL are repeated until Count equals 5. Since the structure is controlled by a boolean expression, the expression may include boolean operands and operators.

Subprogram Example 3-21

```
PROCEDURE MyTwentyFirst;
VAR
    Count : INTEGER;
BEGIN
    Count := 0;
    REPEAT
        WRITELN('Count = ',Count);
        Count := Count + 1;
    UNTIL (Count > 0) AND (Count = 5);
END;
RUN(MyTwentyFirst);
```

Output from Subprogram 3-21



The **WHILE** structure is also a conditional loop, but its condition is checked prior to entering the loop, instead of on exit. The loop's action is not executed at all if the entry condition is not met.

In the first part of the **WHILE** structure, a condition is stated as a boolean expression. It determines whether or not the loop will be entered and when the loop will terminate.

In a sense, the entry condition serves as an **IF** structure—a boolean expression must be true for the structure to be entered. But since the **WHILE** loop is a structure, the expression is evaluated again after the action is completed. If it is still true, the action gets repeated. If the entry condition has become false, the action is skipped entirely, and the subprogram moves on to the next statement.

The statement controlled by **WHILE ... DO** may be either a single statement or a compound **BEGIN...END** construction. Subprogram Example 3-22 shows a **WHILE...DO** structure.

Subprogram Example 3-22

```
PROCEDURE MyTwentySecond;  
VAR  
    Count : INTEGER;  
BEGIN  
    Count := 0;  
    WHILE (Count < 5) AND (Count >= 0) DO  
        BEGIN  
            WRITELN('Count = ',Count);  
            Count := Count + 1;  
        END;  
    END;  
END;  
RUN(MyTwentySecond);
```

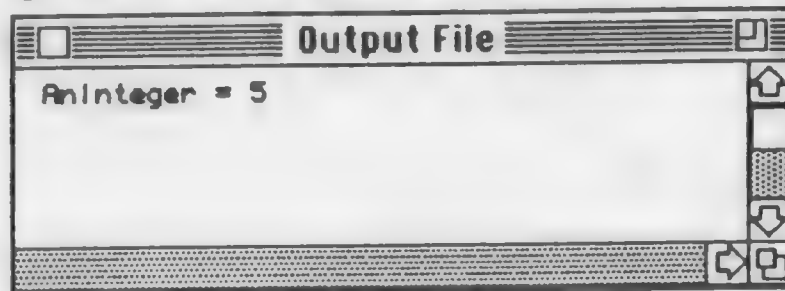
While Count is greater than or equal to zero and less than 5, the compound statement is executed. However, when Count is equal to 5, the loop is terminated. The output from Subprogram Example 3-22 is the same as the output from Subprogram Example 3-21.

The final type of statements that we will look at are **control statements**. Control statements allow the programmer to have direct control over what portions of a subprogram should be executed. This is accomplished through the use of **GOTO** statements.

Subprogram Example 3-23

```
PROCEDURE MyTwentyThird;  
  LABEL 1,2;  
  VAR  
    AnInteger : INTEGER;  
  BEGIN  
    AnInteger := 5;  
    IF AnInteger = 5  
    THEN  
      GOTO 1;  
    WRITELN('AnInteger <> 5');  
    GOTO 2;  
  1:WRITELN('AnInteger = 5');  
  2:END;  
  RUN(MyTwentyThird);
```

Output from Subprogram Example 3-23



Subprogram Example 3-23 demonstrates the use of the **GOTO** statement. **GOTO** statements cause the computer to jump to a corresponding numeric value and then to begin execution at that point. In the above example, since **AnInteger** equals five, the computer will jump to the statement labeled "1:" and then execute the **WRITELN** statement. On the other hand, if **AnInteger** did not equal five, then the computer would execute the following **WRITELN** statement and then jump to the **END** state-

ment labeled "2:". Notice the label declaration which is located immediately after the procedure identifier. All label declarations occur prior to both constant and variable declarations. Also, all labels must be integers between 0 and 9999 and they must have a colon symbol proceeding them.

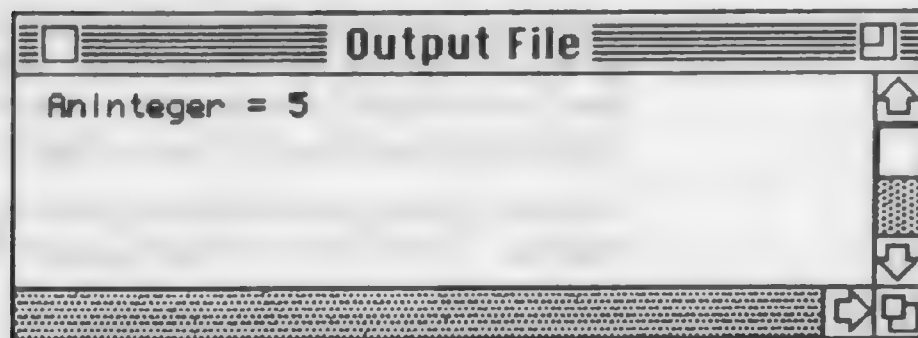
Section 3-5 Procedures and Functions

Internal procedures and functions divide large subprograms into segments that are easier to read, write and understand. We refer to them as internal subroutines because they are created within the main subprogram. Internal subroutines aid the programmer because they allow him to break a large subprogram into smaller units. These smaller units each execute a specific action, thus performing the "divide and conquer" method of accomplishing a large programming task. Subprogram Example 3-24 demonstrates an internal procedure which is called Inner.

Subprogram Example 3-24

```
PROCEDURE MyTwentyFourth;  
VAR  
    AnInteger : INTEGER;  
  
    PROCEDURE Inner(VAR AValue : INTEGER);  
    VAR  
        AnInteger2 : INTEGER;  
    BEGIN  
        AnInteger2 := 4;  
        AValue := AValue + AnInteger2;  
    END;  
  
BEGIN  
    AnInteger := 1;  
    Inner(AnInteger);  
    Writeln('AnInteger = ',AnInteger);  
END;  
RUN(MyTwentyFourth);
```

Output from Subprogram Example 3-24



Procedure Inner performs a specific task, it receives a variable, adds four to it and then returns the variable to the subprogram that called it. Let's look at the execution of the internal subroutine Inner more closely.

Procedure MyTwentyFourth is the main subprogram in the example. Within MyTwentyFourth's statement section, the value of one is assigned to the variable AnInteger. Following this statement, the procedure identifier Inner is executed. Notice that there is a parameter following the identifier. Parameters allow procedures to pass information between each other, just as creation procedure calls such as RECT(0,1,1,0) pass coordinate information through parameters. In Subprogram Example 3-24, only one parameter is passed, AnInteger, which currently has a value of one.

After the procedure call Inner and its parameter has been executed, the computer looks for an internal procedure called Inner within the main subprogram MyTwentyFourth. The computer finds the internal subroutine and checks to make sure that the parameter listed after the Procedure Inner declaration, AValue, has the same data type as the parameter passed by the main subprogram MyTwentyFourth, which was AnInteger. The computer notes that it is going to be returning a value back to the main subprogram MyTwentyFourth since the reserved word VAR is located before the parameter AValue (this process will be discussed shortly).

The computer immediately begins executing the statement section of the internal subroutine Inner. It first assigns four to the variable AnInteger2. It then adds AnInteger2 to the parameter AValue. Upon finding the reserved word END, the computer returns the current value of AValue to the parameter AnInteger. It then continues executing the statements from the main subprogram MyTwentyFourth that follow the statement line which called the internal subroutine Inner. When AnInteger is written out to a text file, its value will be five.

The actions of a procedure are given as procedure declarations. Therefore, in Subprogram Example 3-24, the statements


```

PROCEDURE Inner(VAR AValue : INTEGER);    {The-
VAR                                         Heading}
    AnInteger2 : INTEGER;                {The Block}
BEGIN
    AnInteger2 := 4;
    AValue := AValue + AnInteger2;
END;

```

are defined as the declaration for the procedure Inner. Each procedure declaration consists of a heading followed by a block. The heading specifies the identifier for the particular procedure and its parameters (if any), while the block contains the declaration and statement parts.

As discussed briefly before, parameters are the values that are passed between a procedure and its corresponding declaration. Specifically, parameters which are given with a procedure are referred to as **actual parameters** while those located in its declaration are called **formal parameters**. In reference to Subprogram Example 3-24, the identifier `AnInteger` is an actual parameter while the identifier `AValue` is a formal parameter.

Function declarations are similar to procedure declarations. However, a function declaration defines a part of the subprogram that specifically computes and returns a value. Function declarations may also include arguments (parameters). Arguments that are given with a function declaration are termed **actual arguments** while those listed after the function call are called **formal arguments**. An example of a function declaration is presented in Subprogram Example 3-25.

Subprogram Example 3-25

```

PROCEDURE MyTwentyFifth;
VAR
    AnInteger : INTEGER;

FUNCTION Inner(AValue : INTEGER) : INTEGER;
VAR
    AnInteger, AValue : INTEGER;
BEGIN
    AnInteger := 4;
    AValue := AValue + AnInteger;
    Inner := AValue;
END;

```

```

BEGIN
    AnInteger := 1;
    AnInteger := Inner(AnInteger);
    WRITELN('AnInteger = ',AnInteger);
END;
RUN(MyTwentyFifth);

```

As discussed before, function calls are operands. Therefore, anytime a function call is used (even if it is calling a function that the programmer has written), the function call itself represents a value. Subprogram Example 3-25 performs similar operations as the internal subroutine Inner in Subprogram Example 3-24. A value is passed to the function declaration, the statements of the function add four to the received value and then the new value is assigned to the function identifier. The difference between the two subprogram examples is that the value from Subprogram Example 3-25 is returned in the function's identifier as opposed to being passed through a parameter. Thus, the fundamental difference between a procedure and a function is that a procedure cannot represent a value. Its name in a subprogram merely tells the computer to execute the procedure's statements in its declaration. A function on the other hand, *must* represent a value.

As with a procedure call, the statements to be executed when a function call is encountered are specified by the statement part of a function's declaration. This statement section should contain at least one assignment statement that assigns a value to the function identifier. The result of the function is the last value assigned. If no such assignment statements exists or if it exists but is not executed, the value returned by the function is unspecified.

Procedure and function declarations are located after the declaration section of the main subprogram. Therefore the format of a subprogram which contains a procedure or function would look like the following:

```

Main subprogram heading
Main subprogram definition part
Main subprogram declaration part
    Internal subroutine heading
    Internal subroutine definition part
    Internal subroutine declaration part
    Internal subroutine statement part
Main subprogram statement part

```

As our use of functions and procedures increases, the understanding of local and global identifiers is vital. From the procedure and function examples above, we can see that variables are used not only in a main subprogram, but also in internal routines which are called by the main subprogram. In fact, these variables may even have the same identifier. What effect does changing the value of a variable in a called subprogram have on a variable with the same identifier in the main subprogram? This question will be answered as we explain local and global identifiers.

Identifiers that are defined or declared in the main subprogram are global identifiers. They can be used everywhere: in the main subprogram, in a called procedure, or even a procedure that is declared within a procedure.

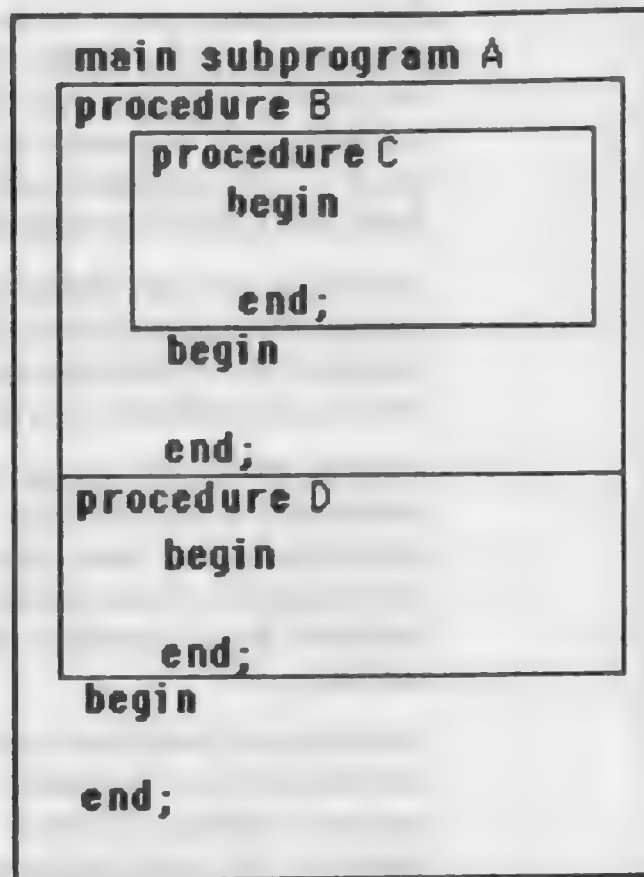
Identifiers which are given meaning within a called subroutine are local identifiers. A local identifier has no meaning in the main program, or in different called subprograms. It can only be used in the subprogram it is declared in (or in subroutines declared within that subroutine).

An internal procedure creates temporary, strictly local use variables and constants that only exist while the procedure is being executed. If a procedure is called five times by the main subprogram, its local variables, constants, and internal routines (if any) are created and disposed of an equal number of times. Therefore, when the function `Inner` is called in Subprogram Example 3-25, the variables created — `AValue` and `AnInteger`, have no effect upon the main subprogram, regardless of the fact that there is a global variable named `AnInteger`. This discussion brings us to the concept of scope.

The word *scope* describes the realm of a variable, constant, or procedure identifier. The scope of an identifier is the portion of a subprogram—called a *block*—in which it can be used to represent a value or action. A block, as just mentioned, consists of a definition part, declaration part, and statement part. When we think in terms of blocks, there is no real distinction between a main subprogram and a called subroutine.

In the following illustration, each block is shown as a box. The scope of a global variable or constant is the entire main subprogram - the largest block. A local identifier's scope is limited to the block it is declared in-its subrou-

tine and other subroutines declared within the subrou-
tine. The illustration demonstrates the scope of con-
stants, variables and internal routines within particular
blocks.



Identifiers (internal routines, variables and constants) defined in each of the subprograms shown in the above example have their scope in the specified blocks.

<u>Identifier Declaration Location</u>	<u>Block Scope</u>
main subprogram A	A, B, C, D
subroutine B	B, C
subroutine C	C
subroutine D	D

Internal routine identifiers—the names of procedures and functions—also obey the rules of scope. In the scope illustration, procedure C may not be called directly from the main subprogram because it is a local procedure. However, procedure B can call procedure C because C is declared within B.

One internal routine may call another internal routine declared before it in the same block. Procedure D may call procedure B, however, they may not call each other in the reverse order. This occurs because the compiler learns about the subprogram by reading it just as a person does. Consequently, the computer can't jump ahead to look for an identifier declaration.

In our description of functions and procedures we referred to the use of parameters and arguments. As mentioned before, arguments are the parameters of functions. Therefore, the proceeding description of parameters equally applies to arguments.

Parameters are composed of **value-parameters** and **variable-parameters**. In different ways these parameters are used to transmit information in and out of procedures and functions.

Value-parameters provide a form of procedure and function input. A value-parameter is a new variable, complete with an identifier and its own storage space in the computer's memory. When a procedure is called, the computer assigns values to the appropriate value-parameters. When the procedure is left behind, the value-parameter storage spaces are taken back by the computer, and the value-parameters cease to exist. Thus, a value-parameter is a variable that is local to a subprogram. Its starting value is passed to it as a parameter of the procedure call. Since a value-parameter is a local variable, changing its value has no effect on the value of a like-named global variable.

Variable-parameters, on the other hand, provide procedure output. No new variables are created. Instead, for the duration of the procedure, a memory space that had previously been known by just one identifier—the name of a global variable—is known by another name as well—the variable-parameter identifier. The two identifiers may even be identical. An assignment to the new identifier, like an assignment to the old global identifier, changes the stored value.

In order to differentiate between the two types of parameters in a formal parameter list, variable-parameters are preceded by the reserved word **VAR**. Parameters without a preceding **VAR** are value-parameters.

Parameter list compatibility is required of the parameter lists of corresponding formal and actual procedural or

functional parameters. Two parameter lists are compatible if they contain the same number of parameters and if the parameters in corresponding positions are of identical type.

Subprogram Example 3-26 demonstrates the use of value and variable parameters.

Subprogram Example 3-26

```
PROCEDURE MyTwentySixth;
VAR
  value1, value2, value3 : INTEGER;
  average : REAL;

  PROCEDURE Inner(value1,value2,value3:INTEGER;
                  VAR average:REAL);
  VAR
    sum : INTEGER;
  BEGIN
    sum := value1 + value2 + value3;
    average := sum/3;
    value1 := 2;
    value2 := 9;
    value3 := 4;
  END;

BEGIN
  value1 := 1;
  value2 := 3;
  value3 := 7;
  average := 0;
  Inner(value1,value2,value3,average);
  WRITELN(value1,' ',value2,' ',value3,' ',average);
END;
RUN(MyTwentySixth);
```

Procedure MyTwentySixth has three variables of type *integer*-value1, value2, value3, and one of type *real*-average. It assigns each variable a value in its statement section (value1 := 1, value2 := 3, value3 := 7, average := 0). Procedure Inner is then called and four parameters are included with the call.

Procedure Inner receives the four parameters and designates value1, value2 and value3 as value parameters and average as a variable parameter. Procedure Inner also declares its own local variable-sum. The variable sum is assigned the value of adding value1, value2 and value3. The variable average is then assigned the result of sum

divided by three. Procedure Inner's statement section then assigns variable value1 the value of 2, variable value2 the value of 9 and variable value3 the value of 4. Procedure Inner then comes to an end and execution of Procedure MyTwentySixth continues precisely after the procedure call statement by writing out the values of value1, value2, value3 and average.

The question at this point is what values will be written out in the WRITELN statement. If you guessed 2, 9, 4 and 3.67, re-read the manual concerning value and variable parameters. Remember that value parameters cannot be changed by a subroutine, even if assignment statements exist in the subroutine to change them. Therefore, the values that will actually be written out will be 1, 3, 7, 3.67.

Unit IV **Importing and Exporting Information in MiniPascal**

Introduction This unit describes the use of MiniPascal files, including the declaration of files in a subprogram. It also discusses several of the pre-defined procedures for importing and exporting information.

Section 4-1 **Using Text Files in MiniPascal**

Text files are a medium for storing information. This information may be in the form of a file format, a subprogram, or the results from the execution of a subprogram. Text files are versatile in the fact that they can be opened by any word processing program and thus manipulated in a specific fashion.

In order to open and close specific files for input or output, each file must be declared as a file variable. A file variable is merely an identifier which is declared to be of type *text*. Subprogram Example 4-1 shows several file variables.

Subprogram Example 4-1

```
PROCEDURE FileDemo;
VAR
    file1, file2 : TEXT;
    s : STRING;
BEGIN
    file1 := 'File #1';
    file2 := 'File #2';
    Rewrite(file1);
    Writeln('Output to file1. ');
    Close(file1);
    Open(file2);
    Read(s);
    Close(file2);
END;
RUN(FileDemo);
```

As shown in previous examples, the default output file is entitled, 'Output File.' If no file routines are executed, all information will be written to this file.

In Subprogram Example 4-1, two file variables have been declared, file1 and file2. The first file routine in the example is a procedure call. This call, REWRITE, tells the computer that all future output information will be stored in file1, which has the file name "File #1." If no text file exists which is called "File #1," the computer will

create a new text file and name it accordingly. However, if a text file does exist by that name, the computer will write over the contents of the previous file. Thus, the procedure REWRITE either creates a new file for output or opens an already existing text file. When this routine is called, all output will be written to the specified file.

The procedure CLOSE is also called from Subprogram Example 4-1. The procedure CLOSE closes a specified open file. Once this procedure has been called, output can no longer be written to the specified file unless it is reopened.

As shown in Subprogram Example 4-1, only one export file may be opened at one time. Thus, if the programmer wishes to export information to a file different from the currently opened file, he must first close the current file and then open the new file.

Text files may also be used to store information which can be read into a MiniPascal subprogram. Prior to reading information from a text file, the file must be opened. One of the routines which opens a text file for reading is Procedure Open. If no text file exists, a MiniPascal error is generated. Once a text file is opened for reading, pre-defined routines exist which read information from a text file. One of these routines is Read, which imports data from the file into a MiniPascal variable. Files which are opened for reading must also be closed. Thus, Procedure Close must be used to close the import file. As with export files, there can only be one import file open at one time. The programmer must close a previously opened import file prior to opening another file.

The above file routines, as well as others, are discussed in detail in Unit V.

Unit V **MiniPascal Predefined Routines**

Introduction The following unit explains the pre-defined MiniPascal routines. These routines may be implemented as described in Section 1-1, "Getting Started." This unit contains many concepts and terms which are described in Unit I, "Using MiniPascal as a File Format." The reader should become familiar with this material prior to proceeding.

Section 5-1 **Selection Routines**

Selection procedures allow the programmer to select or deselect graphic objects within a MiniCad+ drawing. Two procedures exist for these operations: Procedure SelectAll and Procedure DSelectAll.

Procedure SelectAll

When objects are created using MiniPascal, they are automatically selected by default. An object appears selected by the appearance of "handles" surrounding it (see Diagram 5-1). When using MiniPascal as a file format or a programming language, there may be times when some objects are selected and others are not. When Procedure SelectAll is called, it selects all objects in a MiniCad+ file, this includes objects on different layers. Statement Example 5-1 demonstrates this process:

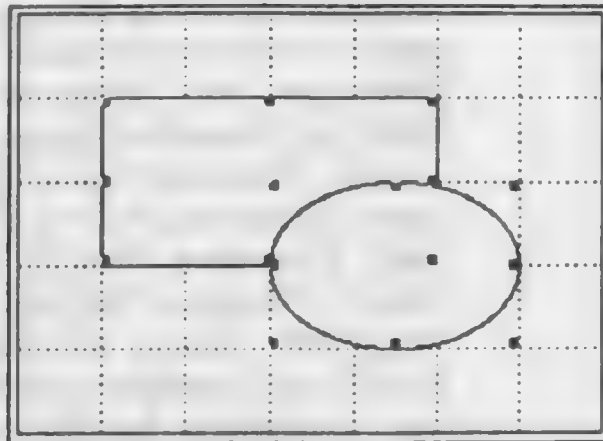
Statement Example 5-1

Procedure SelectAll Example

RECT(-1,1,1,0);	{creates a rectangle}
OVAL(0,0.5,1.5,-0.5);	{creates an oval}
DSELECTALL;	{deselects all objects}
SELECTALL;	{selects all objects}

The statements above produce a rectangle and an oval. Each object is initially selected by default but is deselected by the Procedure DSelectAll (we will examine this procedure next). The Procedure SelectAll then selects all objects (as shown in Diagram 5-1).

Diagram 5-1
Selected Objects



Procedure DSelectAll

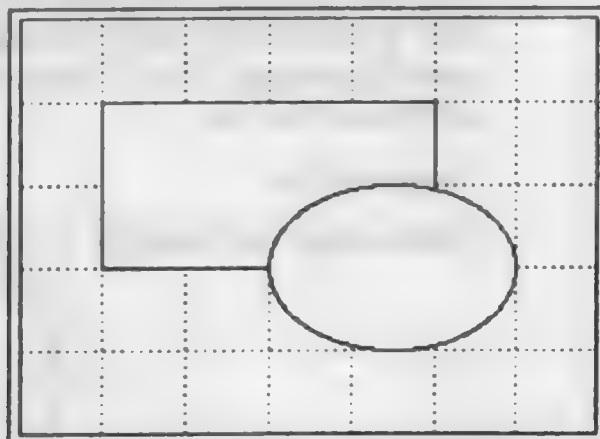
As already described, the Procedure DSelectAll deselects all objects which are selected. Each object's handles disappear. Statement Example 5-2 demonstrates this procedure by creating a rectangle and an oval (which are automatically selected by default) and then deselecting them. The output from the statements is exhibited in Diagram 5-2.

Statement Example 5-2

Procedure DSelectAll Example

RECT(-1,1,1,0);	{creates a rectangle}
OVAL(0,0.5,1.5,-0.5);	{creates an oval}
DSELECTALL;	{deselects all objects}

Diagram 5-2
Deselected Objects



Section 5-2 Creation Routines:

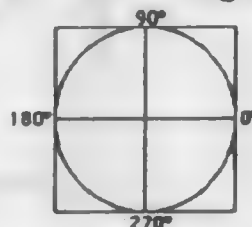
Creation procedures include a collection of powerful routines that allow the programmer to create graphic objects within MiniPascal. These procedures include: Procedure Arc, Procedure DimArcText, Procedure DimText, Procedure Layer, Procedure Line, Procedure LineTo, Procedure Locus, Procedure Move, Procedure MoveTo, Procedure Oval, Procedure PenLoc, Procedure Poly, Procedure Poly3D, Procedure Rect, Procedure RRect, Procedure BeginGroup, Procedure EndGroup, Procedure Group, Procedure UnGroup, Procedure SprdSheet, Procedure SprdAlign, Procedure SprdBorder, Procedure SprdFormat, Procedure SprdWidth, Procedure LoadCell, Procedure Symbol, Procedure SymLocus, Procedure BeginSym, Procedure EndSym, Procedure BeginFolder, Procedure EndFolder, Procedure TextOrigin, Procedure BeginText, Procedure EndText, Procedure BeginMesh, Procedure EndMesh, Procedure BeginXtrd, Procedure EndXtrd, Procedure BeginMXtrd, Procedure EndMXtrd, Procedure BeginSweep, Procedure EndSweep, Procedure BeginPoly, Procedure AddPoint, Procedure EndPoly, Procedure Message and Procedure ClrMessage.

**Procedure Arc(X_1, Y_1, X_2, Y_2 ,
#StartAngle, #ArcAngle : REAL)**

Procedure Arc creates circular arcs of any angle. It draws an arc of the oval that fits inside the rectangle specified by the coordinates (X_1, Y_1, X_2, Y_2). Parameter StartAngle indicates where the arc begins and has a range between 0 and 360. Parameter ArcAngle defines the extent of the arc and also has a range between 0 and 360. The degrees of each angle parameter correspond to the mathematical degrees of a cartesian plane. Diagram 5-3 presents the degrees relative to a rectangle. Statement Example 5-3 and Diagram 5-4 demonstrate the use of Procedure Arc.

Diagram 5-3

Angle Degrees Relative to a Rectangle



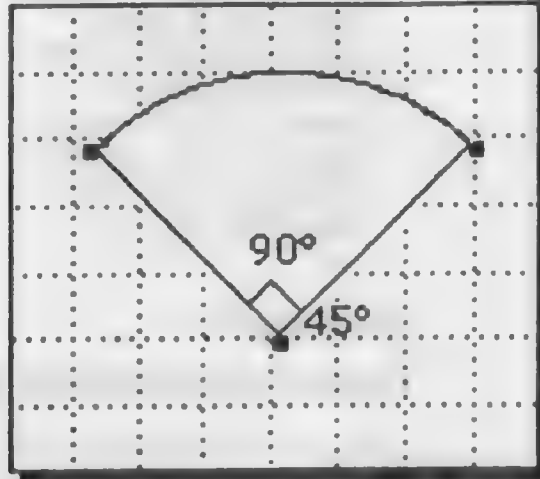
Statement Example 5-3

Procedure Arc

```
ARC(0,2,2,0,#45,#90);
```

Diagram 5-4

Output from Statement Example 5-3



Procedure DimArcText(X,Y : REAL) and Procedure DimText(X,Y : REAL)

The procedures DimArcText and DimText are used for displaying dimension values on screen. When procedure DimArcText is called, an arc dimension appears corresponding to the most recently created arc. The location of the dimension is designated by the parameters (X,Y). Similarly, when procedure DimText is called, a dimension value appears corresponding to the most recently created line. Its location is also designated by the coordinate parameters (X,Y). If the user wishes the dimension text to be located at the default position (the center of the measured arc or line), he may enter the statement DimArcText() or DimText(). Procedures DimArcText and DimText are used primarily in exporting a current Minicad+ file or display to a text file. Statement Example 5-4 and Diagram 5-5 demonstrate these routines.

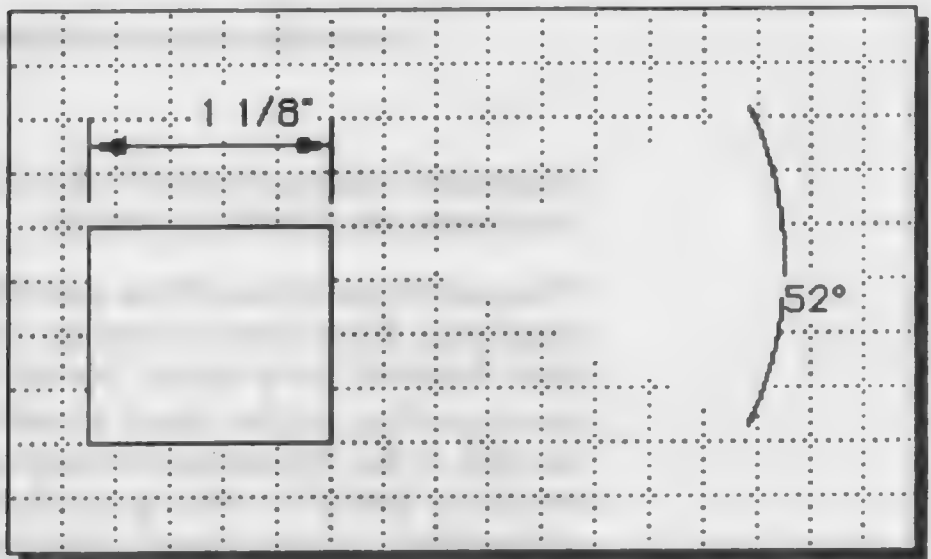
Statement Example 5-4

Procedures DimArcText and DimText

ARC(2 3/4",-2 1/4",6 1/8",-5 5/8",#334,#52);	{Create arc}
DIMARCTEXT(6 1/8",-4");	{Create dimensions}
RECT(2 7/8",-3 3/4",4",-4 3/4");	{Create rectangle}
ARROWHEAD(3);	{Set arrowhead}
MOVETO(2 7/8",-3 3/8");	{Move pen}
LINETO(4",-3 3/8");	{Create line}
DIMTEXT(3 3/8",-3 1/8");	{Create dimensions}
ARROWHEAD(0);	{Reset arrowhead}
MOVETO(2 7/8",-3 1/4");	{Move pen}
LINETO(2 7/8",-3 5/8");	{Create line}
MOVETO(4",-3 1/4");	{Move pen}
LINETO(4",-3 5/8");	{Create line}

Diagram 5-5

Output from Statement Example 5-4



Procedure Layer(LayerName : STRING)

Procedure Layer creates an additional layer in a MiniCad+ file. The layer's name is designated by the parameter LayerName which is of type *string*. When procedure Layer is called, the new layer becomes the active layer. If the parameter LayerName is the name of a currently existing layer, no new layer is created. However, the specified layer becomes the active layer. Statement Example 5-5 demonstrates the use of Procedure Layer.

Statement Example 5-5

Procedure Layer Example

RECT(-1,1,0,0);	{creates a rectangle}
RECT(-1,-0.5,0,-1.5);	{creates a rectangle}
LAYER('LayerTwo');	{creates a new layer}
OVAL(0,1,1,0);	{creates an oval on the new layer}
OVAL(0,-0.5,1,-1.5);	{creates an oval on the new layer}

Two rectangles are created on the current layer. Procedure Layer is called and the new layer is given the string name 'Layer Two.' Two ovals are then created on the new layer.

Procedure Line(dX,dY : REAL) and Procedure LineTo(X,Y : REAL)

Two routines are available for drawing lines. When discussing the creation of graphics, we refer to the drawing tool as the graphics pen and its location on the graphics screen as the pen location. Procedure LineTo(X,Y) draws a line from the current pen position to the coordinate (X,Y). Procedure Line(dX,dY) draws a line from the current pen position to the point (X + dX, Y + dY), where dX and dY are the relative moves in the X and Y directions. Statement Example 5-6 provides an example of these procedures and Diagram 5-6 shows the output.

Statement Example 5-6

Procedures LineTo and Line

MOVETO(1,1);	{moves the graphics pen to coordinate (1,1)}
LINE(2,2);	{creates a line}
MOVETO(1,2.5);	{moves the graphics pen to coordinate (1,2.5)}
LINE(1,1);	{creates a line}

Important Note:

Procedure LineTo produces a line in absolute coordinates regardless of the current point designation setting (such as relative or distance-angle). Likewise, procedure Line produces a line using relative coordinates even if the current point designation setting is absolute.

Diagram 5-6

Examples of Line and LineTo

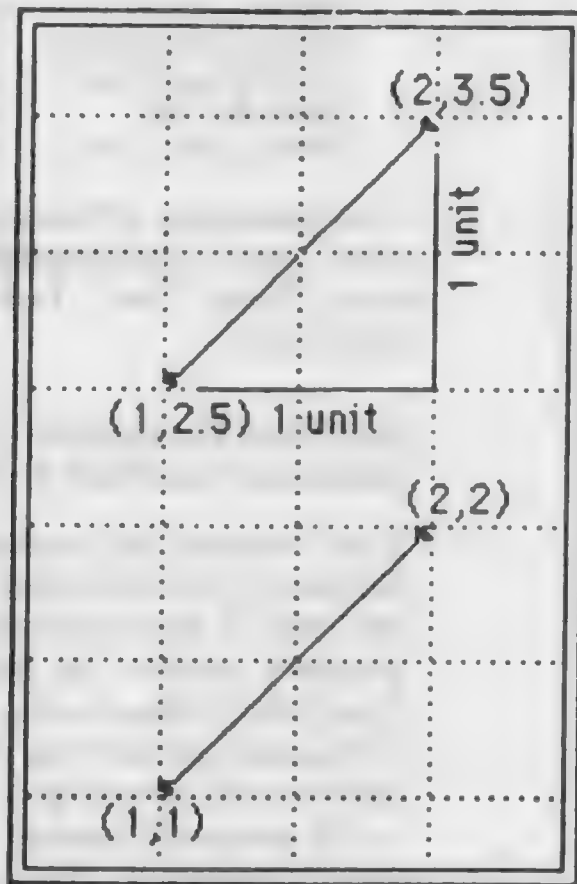


Diagram 5-6 exhibits the fundamental differences between the procedures Line and LineTo. Procedure LineTo draws a line from its starting coordinate (1,1) to the exact coordinate (2,2). Procedure Line draws a line from its starting point (1, 2.5) to a point which is one horizontal unit and one vertical unit away, which happens to be the coordinate (2, 3.5).

Procedure Locus(X,Y : REAL)

Procedure Locus does not create a graphic object, but rather a positioning element. The locus has primarily a 'point of reference' function in MiniCad+. It is used with various functions including Snap to Locus, Align Objects, and three dimensional objects. The parameters X and Y, of type *real*, are coordinate values for its location. Statement Example 5-7 presents this routine.

Statement Example 5-7

Procedure Locus Example

```
LOCUS(1,1); {a locus is created and located at coordinate—  
            (1,1)}
```

Procedure Move(dX,dY : REAL) and Procedure MoveTo(X,Y : REAL)

Procedure MoveTo(X,Y) sets the absolute coordinate position of the graphics pen. Nothing appears on the screen, the pen just moves to the point that we select. Any called object creation routine begins drawing here. Statement Example 5-8 provides an example.

Statement Example 5-8

Procedure MoveTo Example

```
MOVETO(1,1); {Moves cursor to coordinate (1,1)}
```

Procedure Move(dX,dY) causes the cursor to move dX horizontally and dY vertically relative to the current pen position. To move down or to the left, simply use a negative value for the corresponding variable. As with MoveTo, nothing appears on the screen, only the pen moves. Statement Example 5-9 demonstrates this process.

Statement Example 5-9

Procedure Move Example

```
MOVE(3,2); {Moves cursor to 3 units to the right and two  
            units upward}
```

Important Note:

Procedure MoveTo moves the graphics pen in absolute coordinates regardless of the current point designation setting (such as relative or distance-angle). Likewise, Procedure Move moves the graphics pen using relative coordinates even if the current point designation setting is absolute.

Procedure Oval(X_1, Y_1, X_2, Y_2 : REAL)

Procedure Oval creates an oval within a rectangular boundary. The coordinate values represent the upper-left and bottom-right corners of the rectangle. Statement Example 5-10 shows an example of the Oval routine and Diagram 5-7 shows the output from the statement example.

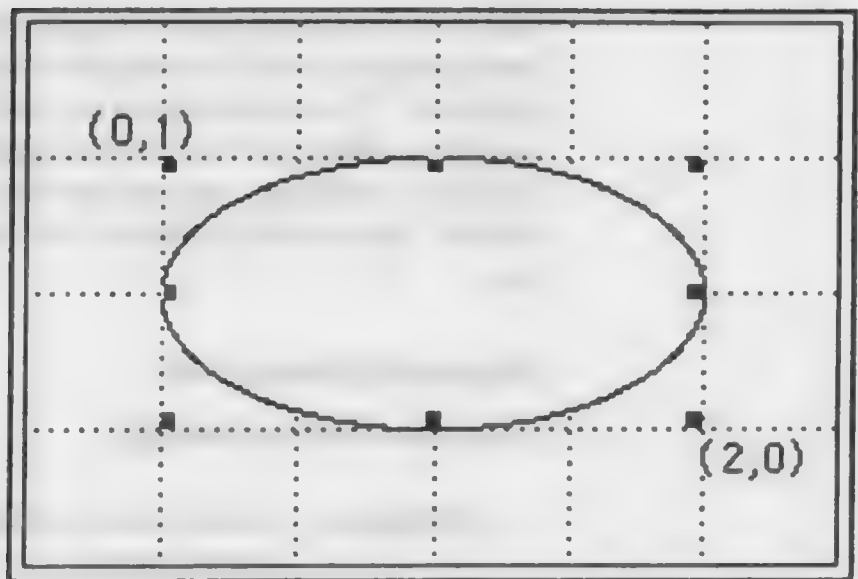
Statement Example 5-10

Procedure Oval Example

```
    OVAL(0,1,2,0);           {creates an oval}
```

Diagram 5-7

Procedure Oval Example



Procedure PenLoc(VAR X,Y : REAL)

Procedure PenLoc returns the current coordinate location of the graphics pen in its parameters X and Y, which are of type *real*. Statement Example 5-11 demonstrates this routine.

Statement Example 5-11

Procedure PenLoc Example

```
PROCEDURE GetPen;
VAR
  X,Y : REAL;
BEGIN
  ABSOLUTE;
  MOVETO(2,2);
  LINETO(1,#45);
  PENLOC(X,Y);
  Writeln('X = ',X:5:2,' Y = ',Y:5:2);
END;
RUN(GetPen);
```

Procedure Poly($X_1, Y_1, X_2, Y_2, \dots, X_n, Y_n$: REAL)

Procedure Poly creates a polygon. Given a set of coordinate points, Procedure Poly will connect them with straight lines in sequence, just like a dot-to-dot drawing. Statement Example 5-12 exhibits Procedure Poly.

Statement Example 5-12

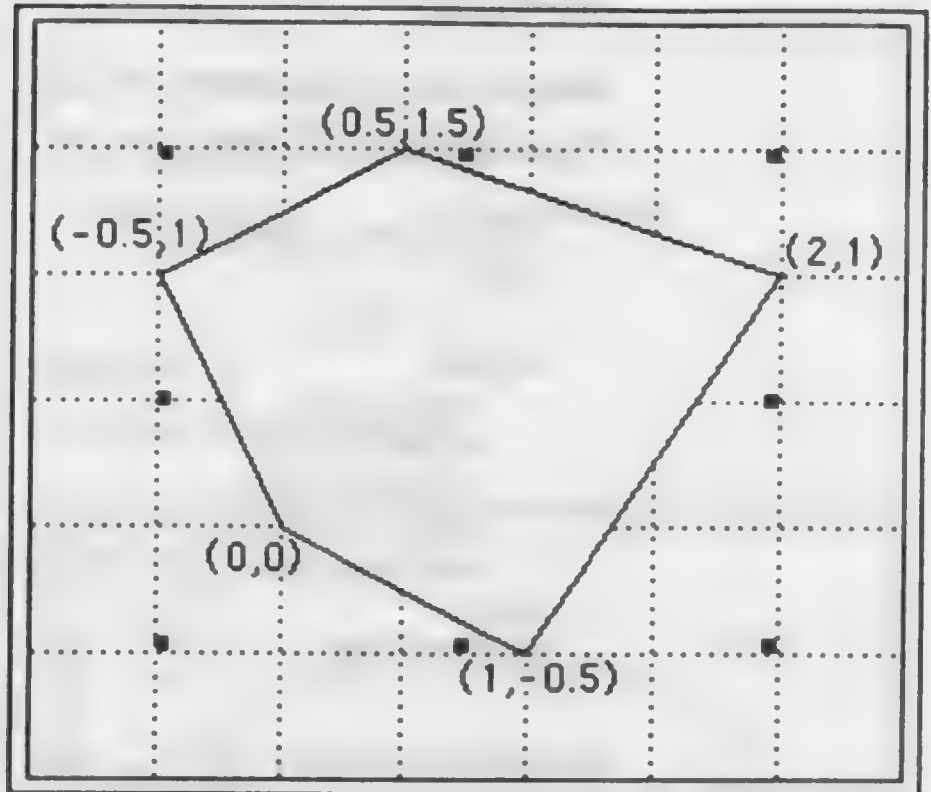
Procedure Poly Example

```
CLOSEPOLY;                                {indicates a closed polygon
POLY(0,0,-0.5,1,0.5,1.5,2,1,1,-0.5);      is going to be created}
                                           {creates a pentagon}
```

Procedure ClosePoly lets the computer know that it should automatically enclose any polygon that is created. This procedure is explained in more detail in the next section. Diagram 5-8 shows the output from Statement Example 5-12.

Diagram 5-8

Procedure Poly Example



Procedure Poly3D($X_1, Y_1, Z_1, \dots, X_n, Y_n, Z_n$: REAL)

Procedure Poly3D creates a three dimensional polygon. Similar to Procedure Poly, Poly3D connects a line between each specified coordinate in space. If the user is viewing a MiniCad+ drawing from a Top View, then the X coordinate specifies horizontal movement (left and right), the Y coordinate specifies vertical movement (up and down) and the z coordinate specifies depth (toward the viewer or away from the viewer). Statement Example 5-13 demonstrates this routine and Diagram 5-9 shows the output.

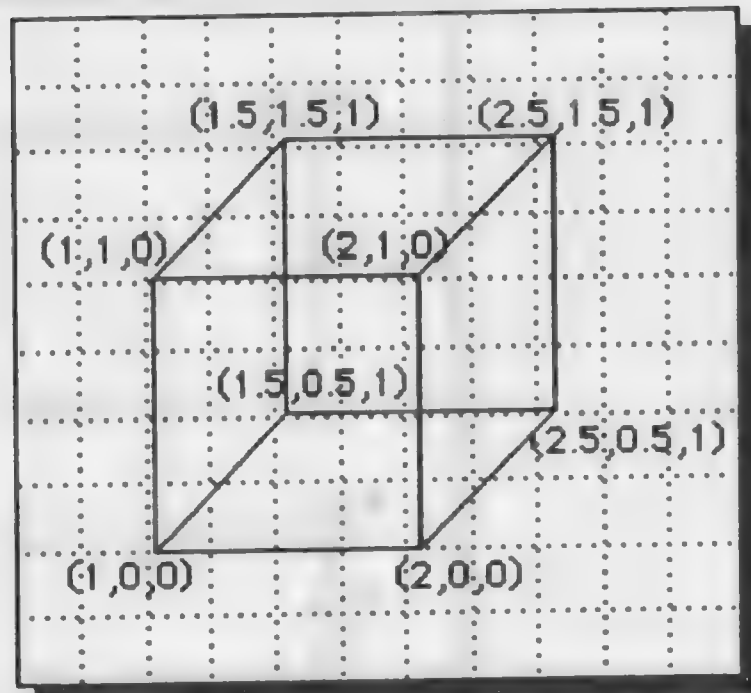
Statement Example 5-13

Procedure Poly3D Example

```
POLY3D(                                     {Creates a three dimensional polygon}  
  1,1,0,  
  1.5,1.5,1,  
  2.5,1.5,1,  
  2,1,0,  
  1,1,0,  
  1,0,0,  
  1.5,0.5,1,  
  1.5,1.5,1,  
  1.5,0.5,1,  
  2.5,0.5,1,  
  2.5,1.5,1,  
  2.5,0.5,1,  
  2,0,0,  
  2,1,0,  
  2,0,0,  
  1,0,0);
```

Diagram 5-9

Procedure Poly3D Example



Procedure Rect(X_1, Y_1, X_2, Y_2 : REAL)

Procedure Rect creates a rectangle given the top-left corner and bottom-right corner coordinates. Statement Example 5-14 demonstrates the routine using coordinate and distance-angle values.

Statement Example 5-14

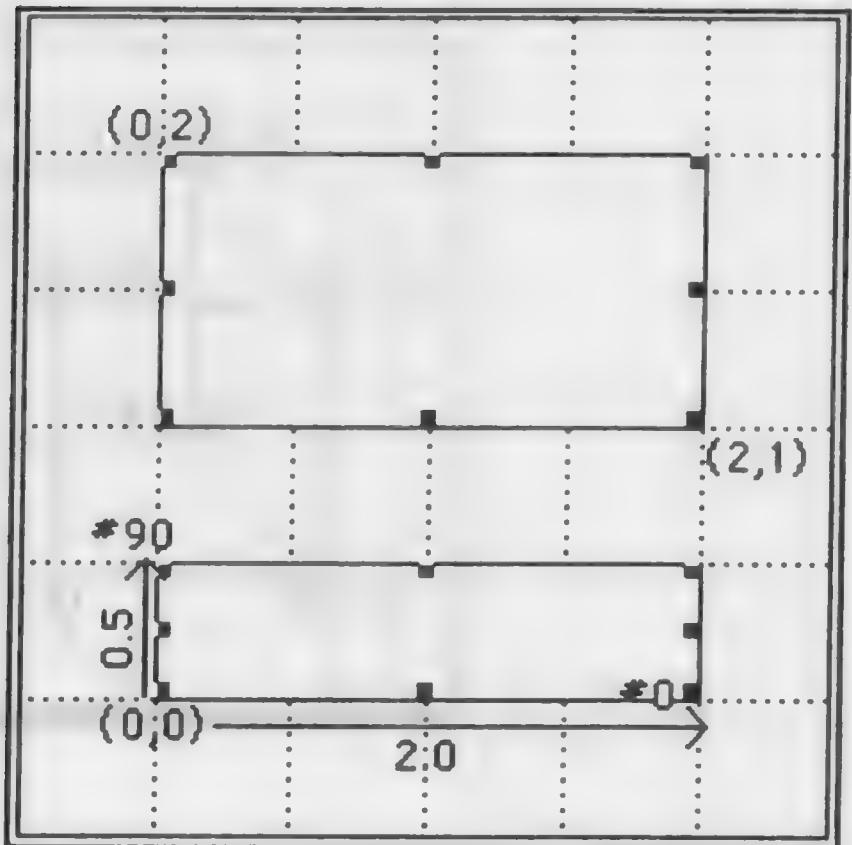
Procedure Rect Example

```
RECT(0,2,2,1);  
RELATIVE;  
RECT(0.5,#90,2,#0);
```

Diagram 5-10 shows the output from Statement Example 5-14.

Diagram 5-10

Rectangles Using Coordinate and Distance-Angle Values



Procedure RRect($X_1, Y_1, X_2, Y_2, XDiam, YDiam$: REAL)

Procedure RRect, which stands for rounded rectangle, creates a rectangle with rounded corners. The routine receives several parameters, two sets of coordinates values and two diameter values, all of type *real*. Statement Example 5-15 presents an example of the Procedure RRect.

Statement Example 5-15

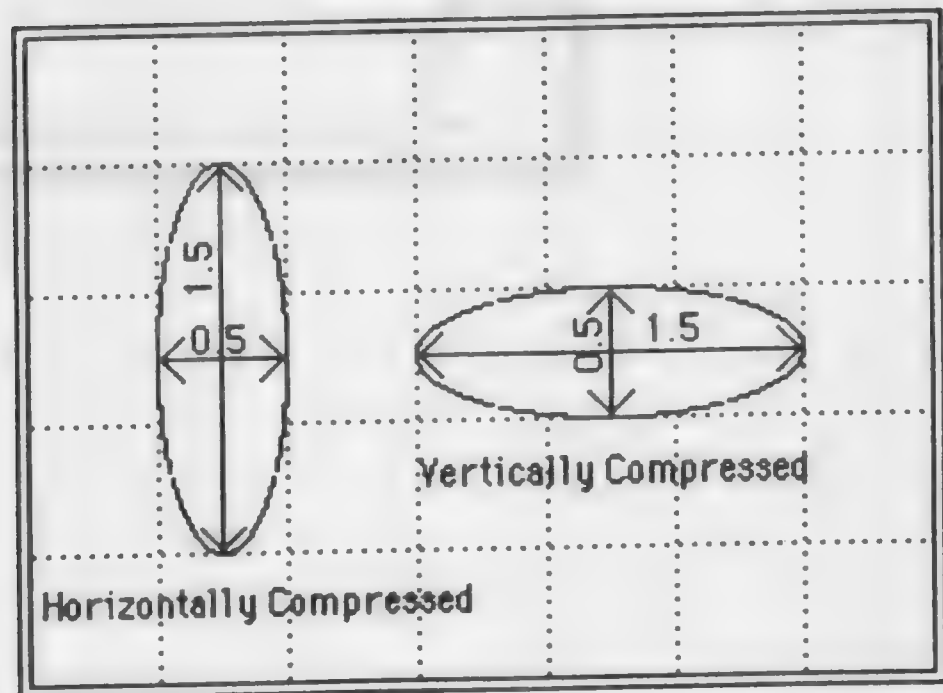
Procedure RRect Example

```
RRECT(-1,0,1.5,-1,1.5,0.5); {creates a rounded rectangle}
```

The X and Y diameters determine the "roundness" of the corners. When working in MiniCad+, you may notice that when you are creating a circle with the mouse, you are able to compress the circle horizontally or vertically, thus creating an oval. In Diagram 5-11, the first oval is compressed horizontally, its x-diameter value equals 0.5 and its y-diameter value equals 1.5. Likewise, the vertically compressed oval has an x-diameter value of 1.5 and a y-diameter value of 0.5.

Diagram 5-11

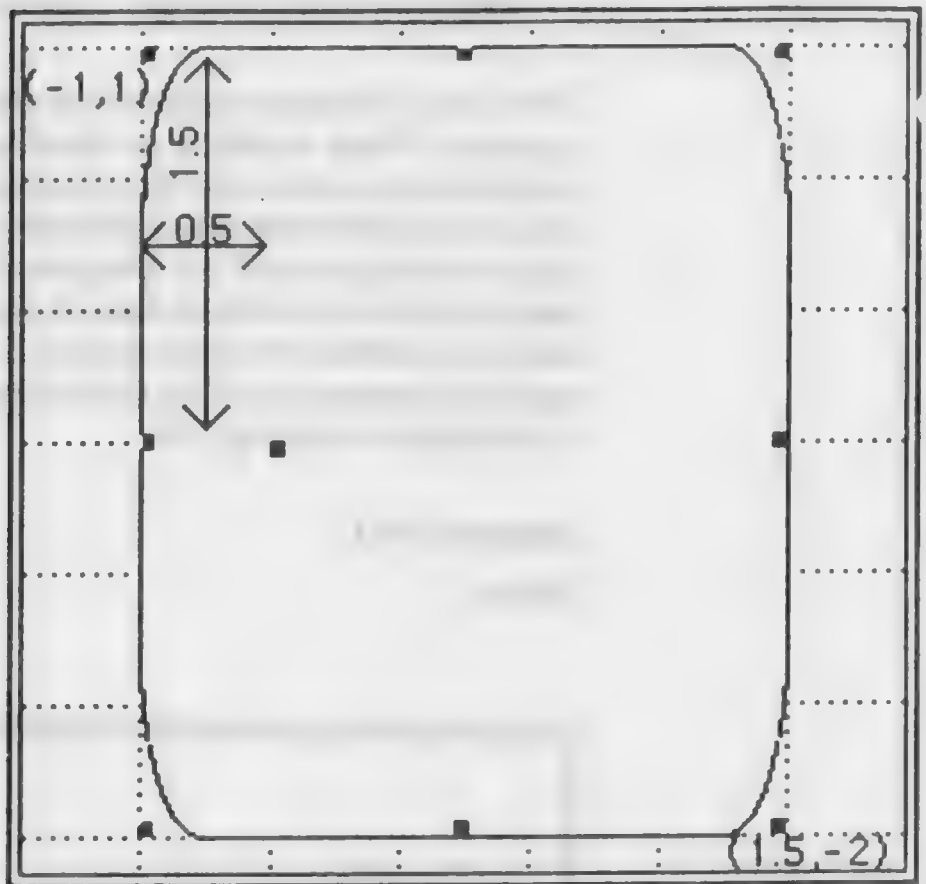
Ovals



When creating a rounded rectangle using MiniPascal, imagine that an oval exists in each corner of the rectangle and that the values of the x and y diameters manipulate each oval as described above. Therefore, the procedure call in Statement Example 5-15 would produce a rectangle with its corners rounded similar to the horizontally compressed oval in Diagram 5-11. Diagram 5-12 shows the output from such a call.

Diagram 5-12

Rounded Rectangle



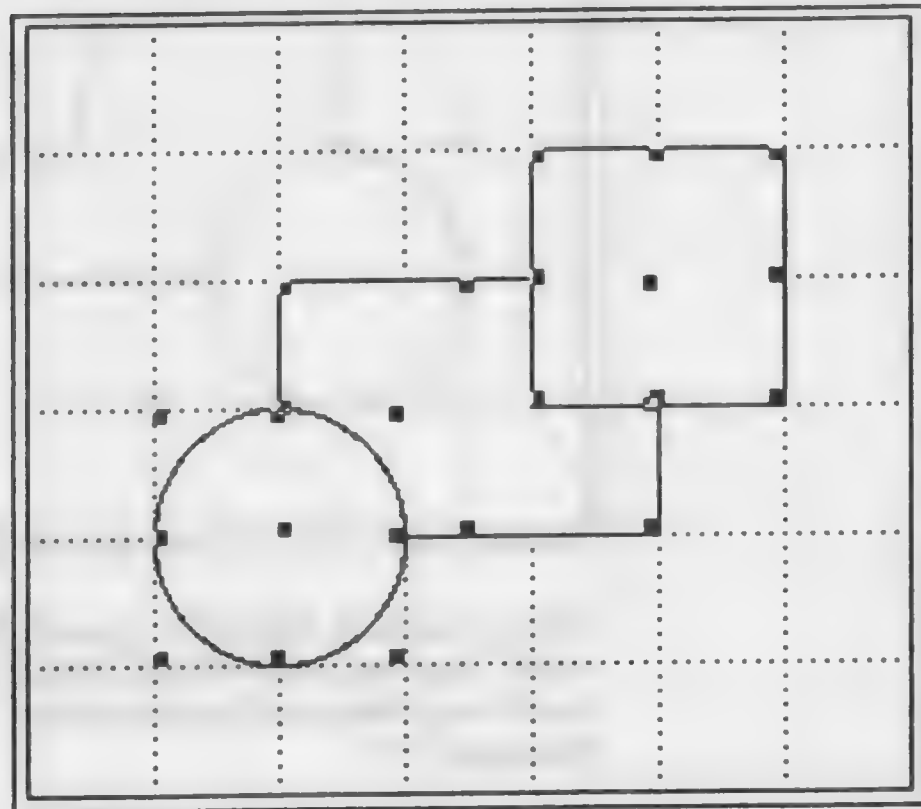
Procedures BeginGroup and EndGroup

The procedures BeginGroup and EndGroup execute the grouping features in MiniPascal. The Procedure BeginGroup initiates the process of combining individual objects and placing them into one collection and the Procedure EndGroup terminates the process.

The grouping process may be thought of as taking several objects and combining them into one object. Several individual objects appear with their own handles (see Diagram 5-13). Grouped objects, however, only have one set of handles among them (see Diagram 5-14). When the BeginGroup routine is executed, a new group collection is initiated and any objects which are created after the BeginGroup statement are included in the collection. This grouping process is terminated by a corresponding EndGroup statement.

Diagram 5-13

Individual Selected Objects



Statement Example 5-16 demonstrates two rectangles and one oval which are created and grouped as one object. The output from the statements is shown in Diagram 5-14.

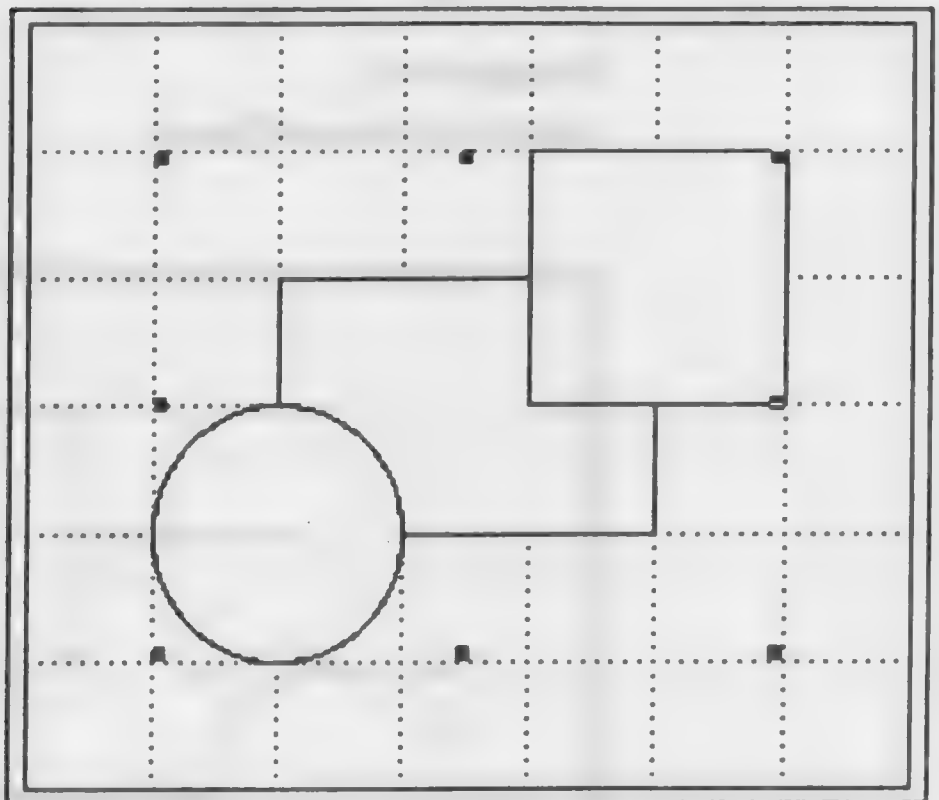
Statement Example 5-16

Procedures BeginGroup and EndGroup Example

BEGINGROUP;	{initiates grouping}
RECT(-1,1,0.5,0);	{creates a rectangle}
RECT(0,1.5,1,0.5);	{creates a rectangle}
OVAL(-1.5,0.5,-0.5,-0.5);	{creates an oval}
ENDGROUP;	{terminates grouping}

Diagram 5-14

Grouped Selected Object



The procedures BeginGroup and EndGroup are nestable. This means that BeginGroup and EndGroup statements may exist within each other. Look at Statement Example 5-17.

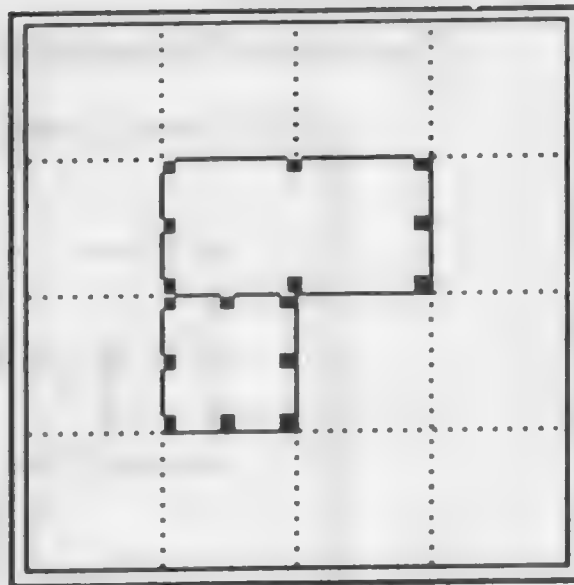
Statement Example 5-17

Nesting Groups Example

BEGINGROUP;	{initiates first grouping}
RECT(-1,1,0,0.5);	{creates a rectangle}
RECT(-1,0.5,-0.5,0);	{creates a rectangle}
BEGINGROUP;	{initiates second grouping}
OVAL(-0.5,0.5,1,0);	{creates an oval}
OVAL(0,0,1,-0.5);	{creates an oval}
ENDGROUP;	{terminates second grouping}
ENDGROUP;	{terminates first grouping}

The first BeginGroup starts the collecting process. At this point the computer is looking for objects to collect. Two rectangles are created. The objects are not grouped as of yet because an EndGroup statement has not been encountered to finalize the process (see Diagram 5-15).

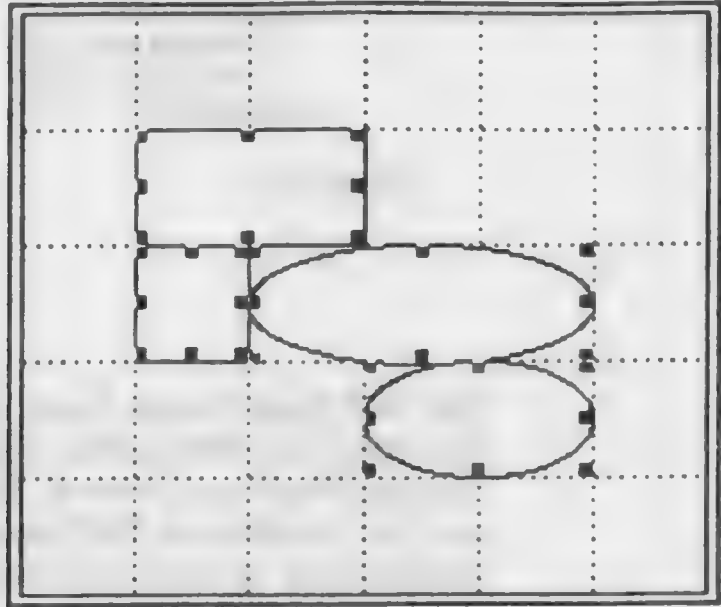
Diagram 5-15
Two Selected Rectangles



Another BeginGroup statement initiates a second grouping process. Two ovals are created but they are not grouped yet either.

Diagram 5-16

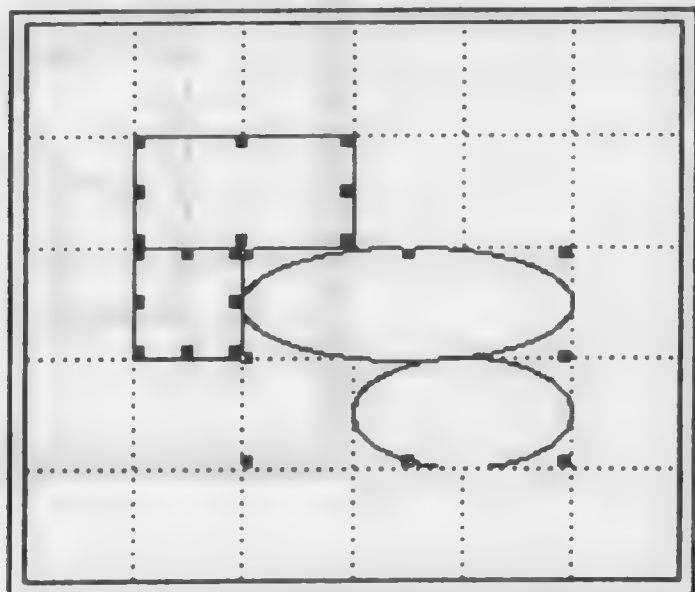
Four Selected Objects



Finally, an EndGroup statement is encountered. The EndGroup statement groups all objects (the two ovals) created between itself and its corresponding BeginGroup statement. This is demonstrated in Diagram 5-17.

Diagram 5-17

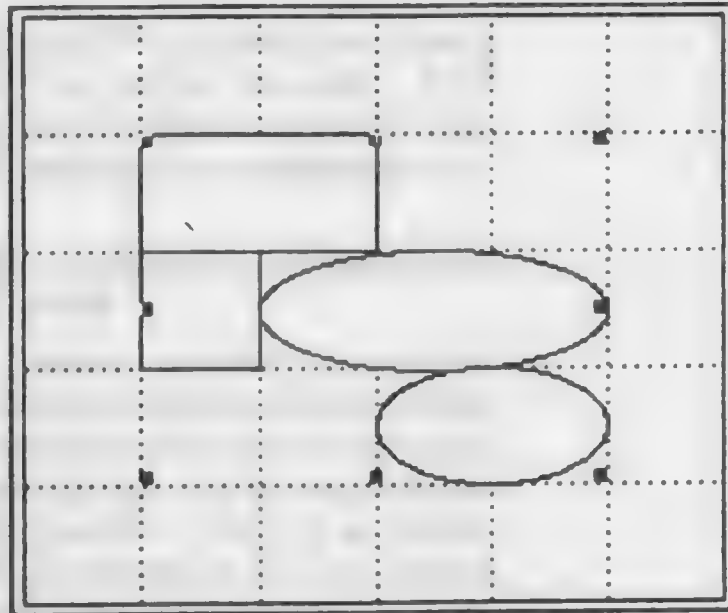
Two Selected Rectangles and One Selected Group



The last EndGroup statement groups the two rectangles and the group of two ovals. This is demonstrated in Diagram 5-18.

Diagram 5-18

One Selected Group



Procedures Group and Ungroup

Procedures Group and Ungroup create or terminate a group of objects. Procedure Group groups all objects selected on the active layer. Procedure Ungroup, on the other hand, breaks a selected group into its individual components.

Statement Example 5-18

Procedure Group Example

RECT(-1,1,0,0);	{creates a rectangle}
RECT(-1,-0.5,0,-1.5);	{creates a rectangle}
GROUP;	{groups selected objects}

Statement Example 5-18, when executed, creates two rectangles. Objects which are created are automatically selected. Thus, when the procedure Group is called, the two rectangles are grouped as one object.

Statement Example 5-19

Procedure UnGroup Example

RECT(-1,1,0,0);	{creates a rectangle}
RECT(-1,-0.5,0,-1.5);	{creates a rectangle}
GROUP;	{groups selected objects}
UNGROUP;	{ungroups selected group}

Statement Example 5-19 is similar to Statement Example 5-18. However, when the Ungroup statement is executed, the previously created group is dispersed and the two rectangles are displayed as individual objects.

Procedure SprdSheet(X,Y : REAL; Rows, Columns : INTEGER)

Procedure SprdSheet creates a spreadsheet with its first cell's top-left corner at coordinate (X,Y). The routine also receives two parameters of type *integer* which specify the dimensions of the spreadsheet in terms of rows and columns of cells. When the spreadsheet is created, MiniCad+ will automatically be placed into its spreadsheet manipulation mode. Statement Example 5-20 and Diagram 5-19 demonstrate this routine.

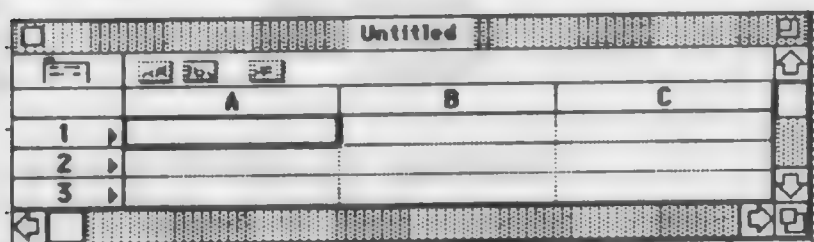
Statement Example 5-20

Procedure SprdSheet Example

SprdSheet(0,0,3,3);	{Spreadsheet with 3 rows and 3 columns}
---------------------	--

Diagram 5-19

A 3 x 3 Worksheet



Procedure SprdAlign(Alignment : INTEGER)

Procedure SprdAlign determines the alignment setting within a spreadsheet cell. The parameter Alignment, of type *integer*, specifies the alignment setting. These alignment settings are presented in Table 5-1. Statement Example 5-21 demonstrates this routine and Diagram 5-20 shows its output.

Table 5-1

Alignment Settings

<i>Alignment Value</i>	<i>Setting</i>
1	General
2	Left
3	Right
4	Center

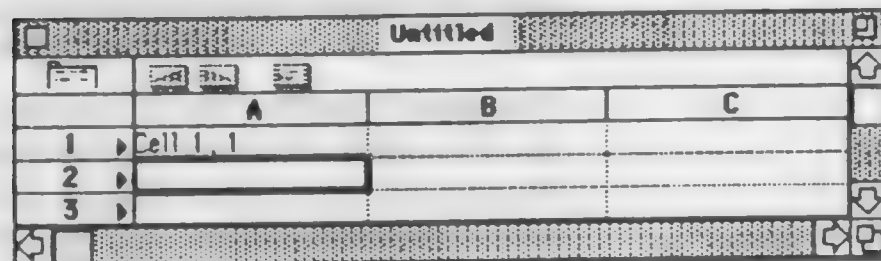
Statement Example 5-21

Procedure SprdAlign Example

```
SprdSheet(0,0,3,3);    {create a spreadsheet}
SprdAlign(2);           {left align the contents of future cells}
LoadCell(3,3,'Cell 1,1') {fill the contents of the first cell}
```

Diagram 5-20

Output from Statement Example 5-21



	A	B	C
1	Cell 1, 1		
2			
3			

Procedure SprdBorder(Top,Left,Bottom,Right : BOOLEAN

Procedure SprdBorder determines the border setting for cells within the active spreadsheet. The parameters Top, Left, Bottom and Right, of type *boolean*, specify the location of the border settings. If the parameter is true, a border is placed at the appropriate location. Statement Example 5-22 demonstrates this routine and Diagram 5-21 shows its output.

Statement Example 5-22

Procedure SprdBorder Example

```
SPRDSHEET(0,0,3,3);
SPRDBORDER(TRUE,FALSE,TRUE,FALSE)
LOADCELL(1,1,'Cell 1,1');
```

Diagram 5-21

Output from Statement Example 5-22

	A	B	C
1	Cell 1, 1		
2			
3			

**Procedure SprdFormat(NumericForm,Accuracy : INTEGER;
Leader,Trailer : STRING)**

Procedure SprdFormat determines the number format for cells within the active spreadsheet. The parameter NumericForm, of type *integer*, specifies the format for numeric representation. The possible values for NumericForm are show in Table 5-2.

Table 5-2**NumericForm Settings**

<i>NumericForm</i>	<i>Setting</i>
0	General
1	Decimal
2	Decimal using commas
3	Scientific
4	Fractional
5	Dimension
6	Angle

The parameter *Accuracy*, of type *integer*, is used with the following *NumericForm* settings: *Decimal*, *Decimal using commas*, *Scientific*, *Fractional* and *Angle*. With the *Decimal*, *Decimal using commas* and *Scientific* settings, the *Accuracy* parameter specifies the number of decimal places to be displayed in the cell. If the *Fractional* setting is selected, the *Accuracy* parameter determines the denominator that fractions are rounded to. Finally, if the *Angle* setting is used, the *Accuracy* parameter selects the angular accuracy to be displayed. The angular accuracy settings are shown in Table 5-3.

Table 5-3**Accuracy Settings using the Angle Numeric Format**

<i>Accuracy</i>	<i>Setting</i>
1	45°
2	45° 20'
3	45° 20' 10"

The parameters *Leader* and *Trailer*, both of type *string*, specify string values which precede or proceed numeric values in a cell. Neither *Leader* nor *Trailer* may exceed eight characters.

Statement Example 5-23 presents the procedure *SprdFormat* and Diagram 5-22 shows its output.

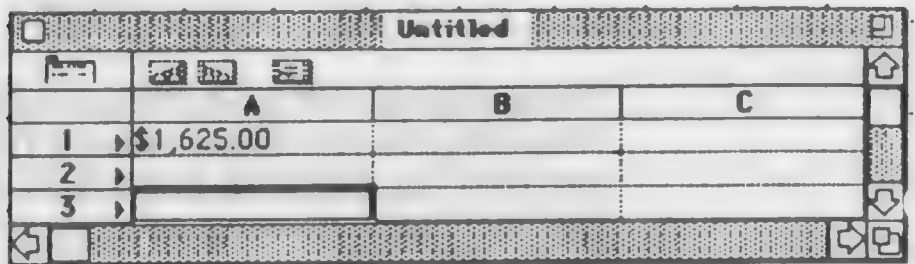
Statement Example 5-23

Procedure SprdFormat Example

```
SprdSheet(0,0,3,3);      {create a spreadsheet}
SprdFormat(2,2,'$','.');  {set the numeric format}
LoadCell(1,1,'=500 * 3.25'); {fill the contents of the first cell}
```

Diagram 5-22

Output from Statement Example 5-23



	A	B	C
1	\$1,625.00		
2			
3			

Procedure SprdWidth(ColumnWidth : INTEGER)

Procedure SprdWidth determines the width of a spreadsheet column. The parameter ColumnWidth, of type *integer*, specifies the width in terms of characters. ColumnWidth must be greater than zero and less than or equal to 255. In order for a column width to be designated, the procedure SprdWidth must be called prior to the LoadCell call for the first cell in the desired column. Statement Example 5-24 demonstrates this routine and Diagram 5-23 shows its output.

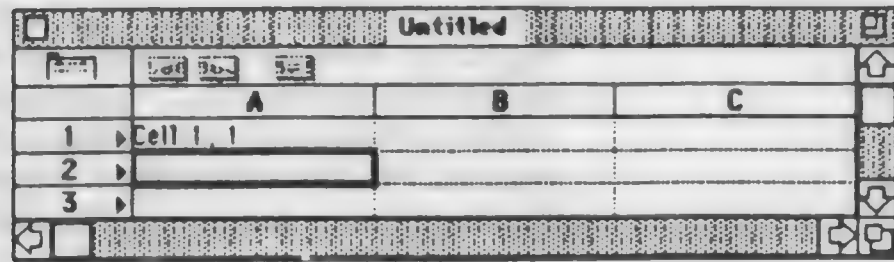
Statement Example 5-24

Procedure SprdWidth Example

```
SprdSheet(0,0,3,3);      {create a spreadsheet}
SprdWidth(15);            {set the column width to 15 characters}
LoadCell(1,1,'Cell 1,1'); {fill the contents of the first cell}
```

Diagram 5-23

Output from Statement Example 5-24



	A	B	C
1	Cell 1, 1		
2			
3			

**Procedure LoadCell(Row,Column : INTEGER;
CellEntry : STRING)**

Procedure LoadCell enters the parameter CellEntry, of type *string*, into the cell designated by the parameters Row and Column, both of type *integer*. Statement Example 5-25 demonstrates procedure LoadCell and Diagram 5-24 shows its output.

Statement Example 5-25

Procedure LoadCell Example

```
SprdSheet(0,0,3,3);           {create a spreadsheet}  
LoadCell(1,1,'= (14 + 2) * 3'); {fill the contents of the first cell}
```

Diagram 5-24

Output from Statement Example 5-25



	A	B	C
1	= (14 + 2) * 3	48	
2			
3			

**Procedure BeginSym(SymbolName : STRING),
Procedure EndSym**

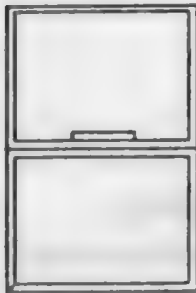
In order to create a symbol and enter it into a MiniCad+ file's symbol library, procedures BeginSym, EndSym and SymLocus must be called. Procedure BeginSym informs the computer that a symbol declaration is going to occur. Its parameter SymbolName, which is of type *string*, specifies the name of the new symbol. Procedure EndSym informs the computer that the symbol declaration has ended. Statement Example 5-26 demonstrates a symbol declaration. Diagram 5-25 shows the appearance of the symbol in the symbol library.

Statement Example 5-26

Procedures BeginSym, EndSym Example

Diagram 5-25

Appearance of
Statement
Example 5-26



```
BEGINSYM('Window');
  Rect(-5'-11",-1",-2'-0",-5'-11");
  Rect(-5'-8 3/4",-2 1/4",-2'-2 1/4",-2'-10");
  Rect(-5'-8 3/4",-3'-1 3/4",-2'-2 1/4",5'-9");
  Rect(-4'-7",-2'-7 3/4",-3'-3 1/2",-2'-9 1/2");
  MoveTo(-5'-11",-3'-0");
  LineTo(-2'-0",-3'-0");
ENDSYM;
```

**Procedure Symbol(SymbolName, : STRING; X,Y,
#Rotation : REAL)**

Procedure Symbol places a symbol library entry into the active layer. The parameter SymbolName, of type *string*, specifies which symbol in the symbol library is to be selected. The coordinates X and Y, of type *real*, determine the location where the selected symbol's center will be placed. The parameter Rotation, of type *real*, specifies the rotation of the symbol when it is placed within the drawing. Statement Example 5-27 demonstrates the placement of the symbol definition Window, which was created in Statement Example 5-26, into a MiniCad+ drawing.

Statement Example 5-27

```
SYMBOL('Window' 3 3029/4096",-4 2069/4096", #0°);
```

Statement Example 5-28

No longer Applicable

Diagram 5-26

No longer Applicable

Procedures BeginFolder and EndFolder

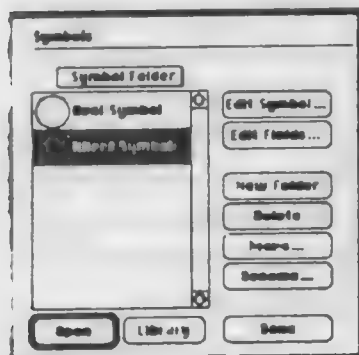
Procedures BeginFolder and EndFolder are routines used to create symbol library folders. The procedure BeginFolder instructs the computer to place all of the following symbols or symbol folders in a specified folder. This folder is named through a NameObject procedure call which will be discussed in the next section. The procedure EndFolder terminates the BeginFolder call. Procedures BeginFolder and EndFolder may only contain symbol declarations. Statement Example 5-29 demonstrates the use of these routines. Diagram 5-27 shows the symbol library entries.

Statement Example 5-29

```
NAMEOBJECT('Symbol Folder');
BEGINFOLDER;
  BEGINSYM('Oval Symbol');
    PENSIZE(14);
    PENPAT(2);
    FILLPAT(1);
    FILLFORE(0,0,0);
    FILLBACK(65535,65535,65535);
    PENFORE(0,0,0);
    PENBACK(65535,65535,65535);
    OVAL(-1/4",1/4",3/4",-3/4");
  ENDSYM;
  BEGINSYM('RRect Symbol');
    RRect(-12'-6",9'-8",-5'-8",4'-6",2'-3 1365/4096",
          1'-81365/2048");
  ENDSYM;
ENDFOLDER;
```

Diagram 5-27

Symbol Folder and
Contents in Symbol
Library

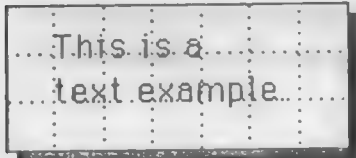


Procedure TextOrigin(X,Y : REAL),

Procedure BeginText and Procedure EndText

Procedures TextOrigin, BeginText and EndText create text objects within a MiniCad+ file. Procedure TextOrigin designates the top-left corner of the text's bounding box by its coordinate parameters (X,Y). Procedure BeginText informs the computer that the following string statement will be the text object's contents. Procedure EndText terminates this process. The text's contents must be a single string value, there cannot be multiple strings. Note that the text's contents may contain a carriage return. Statement Example 5-30 and Diagram 5-28 present these routines.

Diagram 5-28
Statement Example
5-30 Output



Statement Example 5-30

Text Creation Example

TEXTORIGIN(0,0);	{Place the top-left corner at (0,0)}
BEGINTEXT;	{Inform computer to start text}
'This is a text example.'	{Text contents}
ENDTEXT;	{Terminate text creation process}

Procedures BeginMesh and EndMesh

Procedures BeginMesh and EndMesh allow the programmer to create a three-dimensional mesh object. A mesh object allows the user to select and manipulate each vertex within its wire-frame model. The BeginMesh and EndMesh routines are similar to the previously described BeginGroup and EndGroup procedures. Procedure BeginMesh initializes the collecting of graphic objects while procedure EndMesh terminates the process. When procedure EndMesh is encountered, the computer converts any graphic objects created since the last BeginMesh statement into a three-dimensional mesh object. Statement Example 5-31 demonstrates this routine and Diagram 5-29 shows its output.

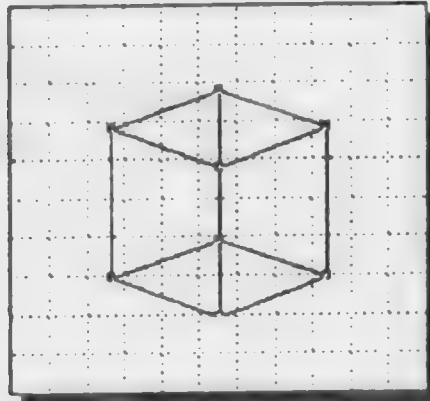
Statement Example 5-31

Procedures BeginMesh and EndMesh

```
BeginMesh;  
  ClosePoly;  
  PenSize(1);  
  PenPat(2);  
  FillPat(0);  
  Poly3D(0',0',4'-0",4'-0",0',4'-0",4'-0",-4'-0",4'-0",0',-4'-  
    0",4'-0");  
  Smooth(0);  
  Poly3D(0',0',0',4'-0",0',0',4'-0",-4'-0",0',0',-4'-0",0');  
  Smooth(0);  
  Poly3D(0',-4'-0",0',0',-4'-0",4'-0",0',0',4'-0",0',0',0');  
  Poly3D(4'-0",-4'-0",0',4'-0",-4'-0",4'-0",0',-4'-0",4'-0",0',-4'-  
    0",0');  
  Poly3D(4'-0",0',0',4'-0",0',4'-0",4'-0",-4'-0",4'-0",4'-0",-4'-  
    0",0');  
  Poly3D(0',0',0',0',0',4'-0",4'-0",0',4'-0",4'-0",0',0');  
EndMesh;
```

Diagram 5-29

Output from Statement Example 5-31

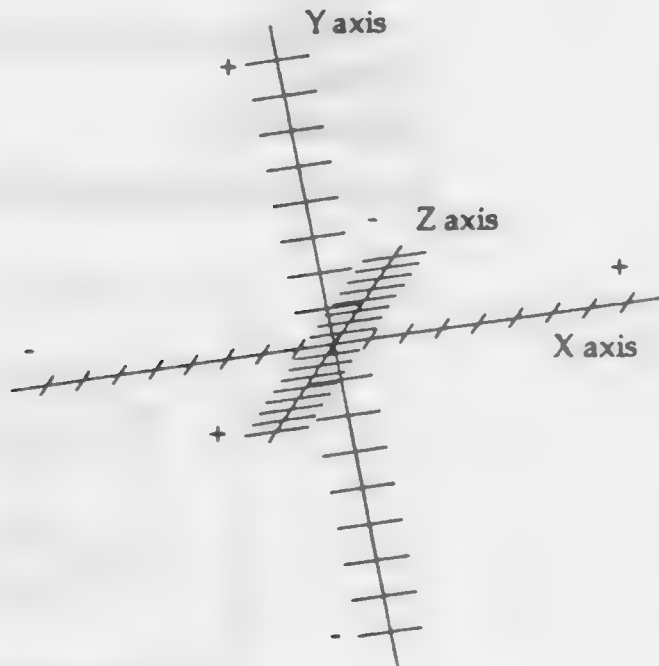


**Procedure BeginXtrd(StartDistance,EndDistance : REAL
and Procedure EndXtrd**

Procedures BeginXtrd and EndXtrd allow the programmer to create a three-dimensional extruded object. An extruded object is a two dimensional object which has been converted into a three-dimensional object by being given depth. The parameters StartDistance and EndDistance, of type *real*, designate the start and end locations of the extrusion. If the programmer is viewing a drawing from the Top View, the extrusion will occur on the Z axis (toward or away from the viewer). Positive StartDistance or EndDistance values occur towards the viewer while negative values move away from the viewer. Diagram 5-30 demonstrates this relationship.

Diagram 5-30

Extrusion values



Procedure BeginXtrd informs the computer that the following graphic objects should be transformed into three-dimensional objects. When Procedure EndXtrd is encountered, the computer extrudes all of the graphic objects created since the last BeginXtrd statement according to the parameters StartDistance and EndDistance. Statement Example 5-32 demonstrates procedures BeginXtrd and EndXtrd while Diagram 5-31 shows its output.

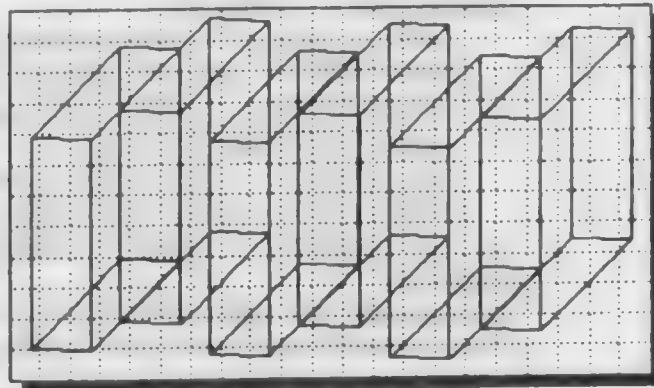
Statement Example 5-32

Procedures BeginXtrd and EndXtrd

```

BeginXtrd(0',4 77/512");                                     {Create extruded →
                                                                object}
    Rect(-1 61/64",125/128",-1 119/256",-375/512");
    Rect(-1 113/512",1 113/512",-375/512",-125/256");
    Rect(-125/256",125/128",0",-375/512");
    Rect(125/128",125/128",1 119/256",-375/512");
    Rect(125/512",1 113/512",375/512",-125/256");
    Rect(1 363/512",1 113/512",2 101/512",-125/256");
EndXtrd;
Rotate3D(#10° 9' 4",#-9° 50' 48",#-1° 45' 14"); {Rotate 3-D object}
Move3D(767/2048",807/2048",-915/4096"); {Move 3-D object}
    
```

Diagram 5-31

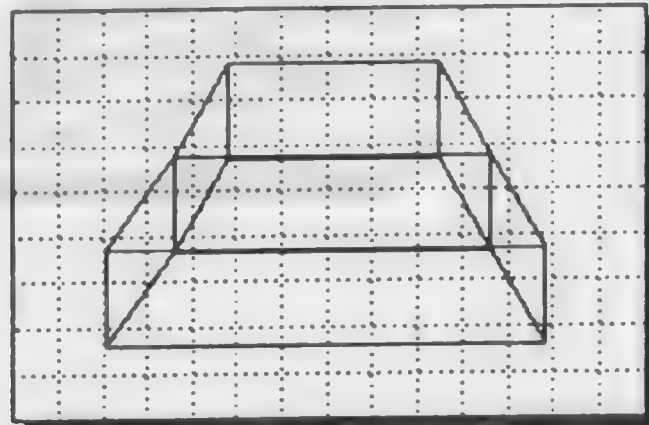


Output from Statement Example 5-32

Procedure BeginMXtrd(StartDistance,EndDistance : REAL) and Procedure EndMXtrd

Procedures BeginMXtrd and EndMXtrd create a multiple extruded object. A multiple extruded object is a combination of two or more extruded objects which are joined together. Diagram 5-32 demonstrates a multiple extruded object.

Diagram 5-32



A multiple extruded object

The parameters StartDistance and EndDistance, of type *real*, designate the start and the end of the extrusion (see Procedures BeginXtrd and EndXtrd for a further explanation). Procedure BeginMXtrd informs the computer that the following graphic objects will be transformed into a multiple extruded object. When Procedure EndMXtrd is encountered, all graphic objects created since the last BeginMXtrd statement are turned into a multiple extruded object. Statement Example 5-33 demonstrates Procedures BeginMXtrd and EndMXtrd and Diagram 5-33 shows its output.

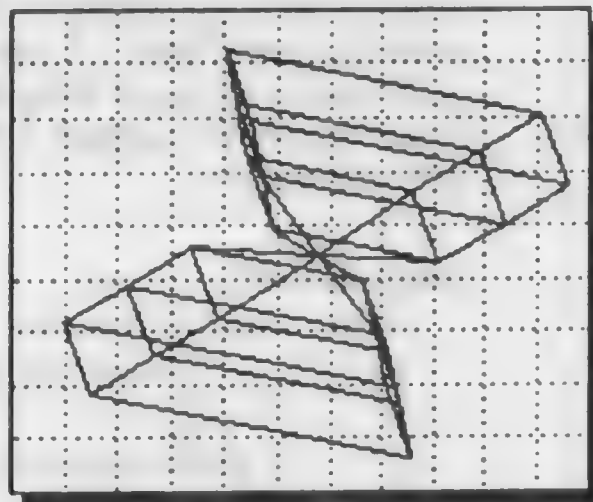
Statement Example 5-33

Procedures BeginMXtrd and EndMXtrd

```
BeginMXtrd(0',1 363/512");
  Rect(-125/128",1 113/512",375/512",375/512");
  Rect(-25/32",1 113/512",275/512",375/512");
  Rect(-75/128",1 113/512",325/1024",375/512");
  Locus(-275/2048",125/128");
  Rect(-75/128",1 113/512",325/1024",375/512");
  Rect(-25/32",1 113/512",275/512",375/512");
  Rect(-125/128",1 113/512",375/512",375/512");
EndMXtrd;
Rotate3D(#52° 47' 45",#-28° 1' 28",#-11° 10' 13");
Move3D(1363/2048",1 75/2048",-2335/4096");
```

Diagram 5-33

Output from Statement Example 5-33



Procedure BeginSweep(#StartAngle, #ArcAngle,
#IncrementAngle, Pitch : REAL)

and Procedure EndSweep

Procedures BeginSweep and EndSweep create a three-dimensional sweep object. A sweep object is a composition of two-dimensional objects which have been duplicated and rotated around some specific location. For example, if the user created a jagged-edged line using the polygon tool and then smoothed it using the Bezier smoothing method, the two dimensional object would look similar to a waved line (see Diagram 5-34). If the object is duplicated and rotated at a specific increment around a center location, and then all of the two-dimensional objects are connected, it would look similar to Diagram 5-35.

Diagram 5-34

Two-dimensional wave line

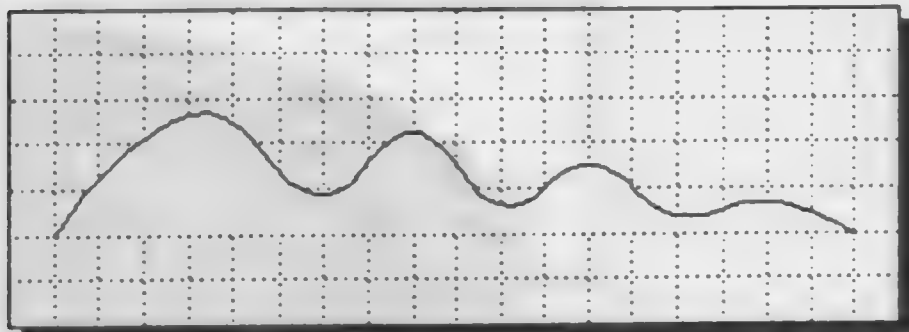
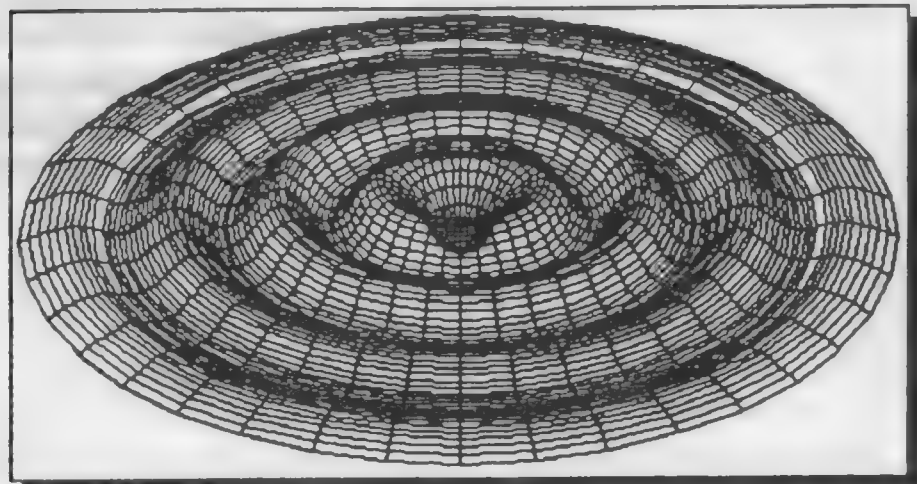


Diagram 5-35

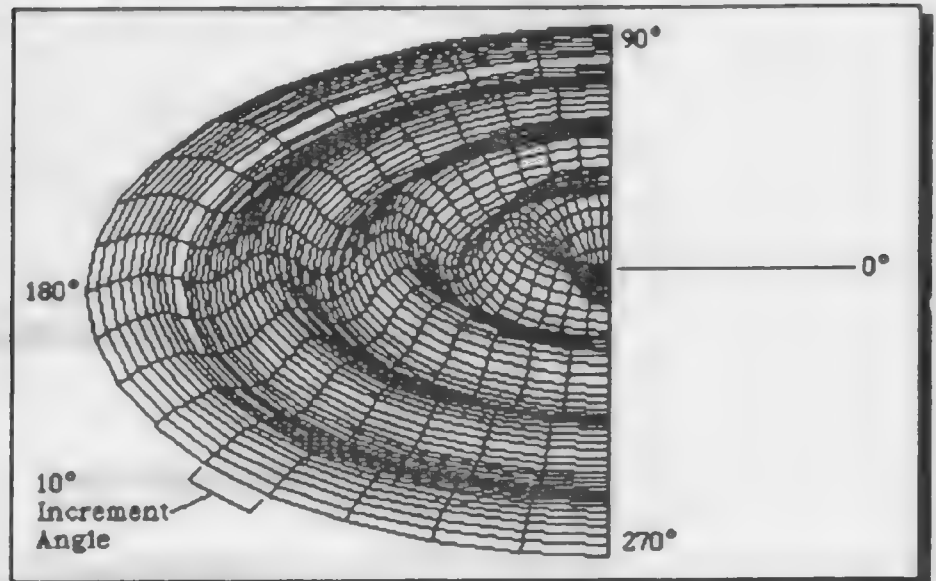
A three-dimensional sweep object



Procedure `BeginSweep` informs the computer that future graphic objects are to be transformed into a three-dimensional sweep object. Procedure `EndSweep` performs the transformation with all graphic objects created since the last `BeginSweep` statement.

Procedure `BeginSweep` receives four parameters. The parameter `StartAngle`, of type *real*, designates the starting location of the sweep while the parameter `ArcAngle`, also of type *real*, determines the degree of sweep. These values are similar to the parameters received by Procedure `Arc` in which the parameter `StartAngle` determines the initial location of the arc and the parameter `ArcAngle` designates the extent of the arc. The above sweep example's `StartAngle` equals 0° and its `ArcAngle` equals 360° . If the programmer entered a `StartAngle` of 90° and an `ArcAngle` of 180° , the sweep object would look like the one shown in Diagram 5-36.

Diagram 5-36
Sweep Object



The parameter `IncrementAngle`, of type *real*, determines the increment between the duplication of objects. Diagram 5-36 shows that the increment between the two bezier smoothed lines is 10° .

Finally, the parameter Pitch, of type *real*, determines the vertical distance that the sweep object will travel for every 360° ArcAngle rotation. This means that as a two-dimensional object is duplicated and rotated, it can also be moved vertically, similar to the steps of a spiral staircase. Thus, if the programmer specifies a pitch which equals zero, while the two-dimensional object is being duplicated and rotated, each duplication remains on the same horizontal level as the previous object (this is shown in the above sweep examples). However, if the programmer enters a pitch greater than or less than zero, each duplication is moved either a positive (up) or negative (down) vertical movement. This vertical movement would be determined by the following equation: vertical movement = pitch * ArcAngle/360. Statement Example 5-34 demonstrates Procedures BeginSweep and EndSweep and Diagram 5-37 show its output.

Statement Example 5-34

Sweep object

```

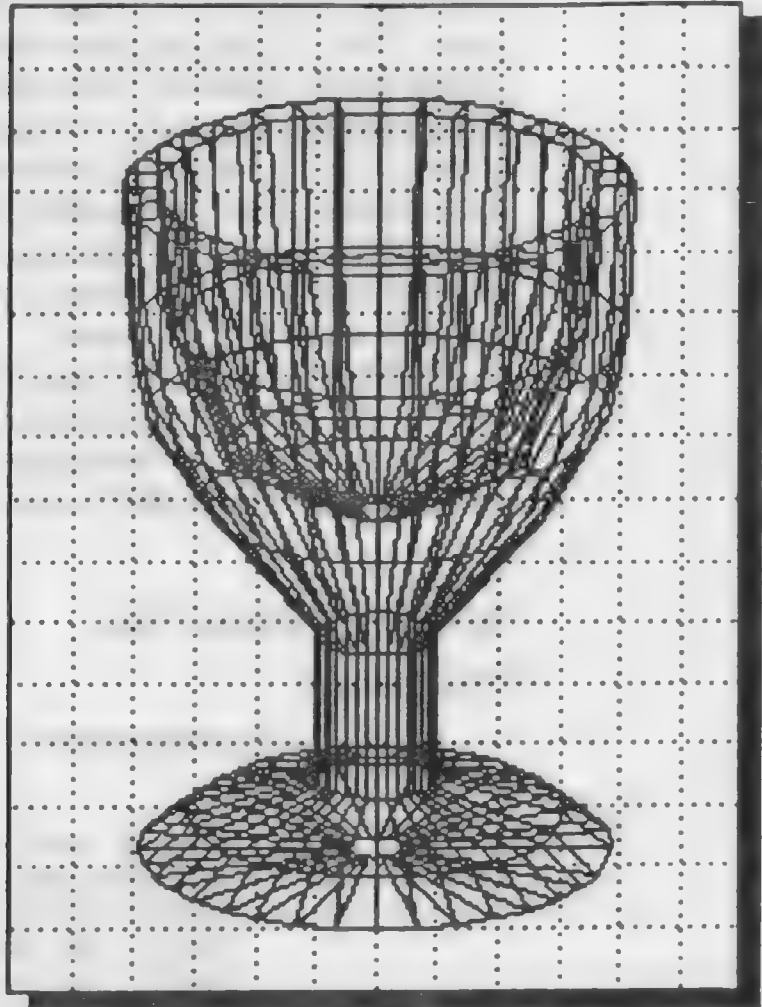
BeginSweep(#0°, #360°, #10°, 0°);           {Create sweep
                                             object}
    Poly(
        -325/512°, -125/1024°,
        -125/1024°, 75/512°,
        25/256°, 675/1024°,
        75/512°, 1 251/1024°,
        75/256°, 1 251/1024°,
        275/1024°, 325/512°,
        25/128°, 75/256°,
        -25/1024°, -75/1024°,
        -125/256°, -325/512°,
        -125/256°, -1 251/1024°,
        25/1024°, -1 401/1024°,
        225/1024°, -1 551/1024°,
        -625/1024°, -1 551/1024°);
    Locus(-375/512°, -225/1024°);
EndSweep;
Rotate3D(#20°, #0°, #0°);                   {Rotate 3D object}
Move3D(0°, 257/4096°, -1459/4096°);         {Move 3D object}

```

Diagram 5-37

Output from Statement Example 5-34.

**Procedure BeginPoly, Procedure AddPoint(X, Y : REAL)
and Procedure EndPoly;**



Procedures **BeginPoly**, **AddPoint** and **EndPoly** allow the programmer to create a polygon and yet perform calculations between creating vertices. (Note: at least two vertices must be created.). Procedure **BeginPoly** informs the computer that a polygon is going to be created, Procedure **AddPoint** adds vertices to the polygon, and Procedure **EndPoly** informs the computer that the polygon object is finished. Subprogram Example 5-1 demonstrates these routines:

Subprogram Example 5-1

```
PROCEDURE PolyTest;
VAR
  x, y : REAL;
BEGIN
  x := 0;
  y := 0;
  BeginPoly;
    AddPoint(x,y);
    x := x + 1;
    y := y + 1;
    AddPoint(x,y);
    x := x + 1;
    y := y - 1;
    AddPoint(x,y);
  EndPoly;
END;
Run(PolyTest);
```

Procedure Message(displayStr : STRING) and Procedure ClrMessage

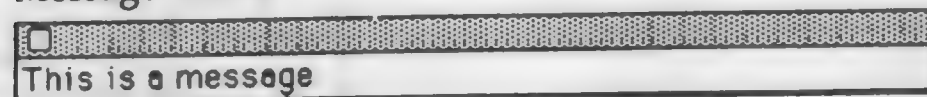
Procedure Message displays the *string* parameter displayStr within a floating message palette while Procedure ClrMessage disposes of the message palette. Only one message palette can be visible at one time. If Procedure Message is called and a message palette already exists, the message displayed is merely updated with the new information. Subprogram Example 5-2 demonstrates procedures Message and ClrMessage. Diagram 5-38 shows the floating message palette.

Subprogram Example 5-2

```
PROCEDURE ShowMessage;
BEGIN
  Message('This is a message');
  Wait(5);
  ClrMessage;
END;
Run(ShowMessage);
```

Diagram 5-38

Message Palette



Attribute Procedures




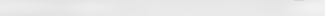

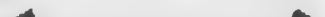

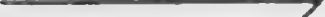
Attribute procedures allow the user to determine the attributes or "characteristics" of created objects in a MiniCad+ file. These routines include: Procedure ArrowHead, Procedure ArrowSize, Procedure ClosePoly, Procedure OpenPoly, Procedure CopyMode, Procedure NameObject, Procedure NameClass, Procedure FillPat, Procedure FillFore, Procedure FillBack, Procedure PenFore, Procedure PenBack, Procedure LckObjs, Procedure UnlckObjs, Procedure PenPat, Procedure PenSize, Procedure Smooth, Procedure ShowLayer, Procedure HideLayer, Procedure GrayLayer, Procedure ShowClass, Procedure HideClass, Procedure NFillClass, Procedure TextFace, Procedure TextFlip, Procedure TextFont, Procedure TextJust, Procedure TextRotate, Procedure TextSize and Procedure TextSpace.

Procedure ArrowHead(ArrowHeadType : INTEGER)

Procedure ArrowHead sets the line marker style. Its parameter, ArrowHeadType, which is of type *integer*, specifies the marker style. The marker styles and their corresponding ArrowHeadType values are listed in Table 5-4. The default ArrowHeadType value is 0. Statement Example 5-35 presents Procedure ArrowHead in a statement form and Diagram 5-39 show its output.

Table 5-4

ArrowHead Types

ArrowHeadType	Marker Style
0	
1	
2	
3	
5	
6	
7	
9	
10	
11	
13	
14	
15	
17	
18	
19	

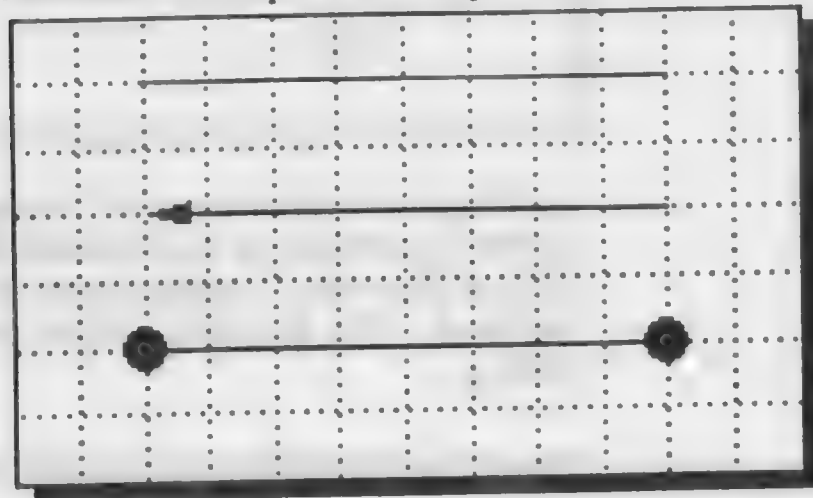
Statement Example 5-35

ArrowHead Example

MOVETO(3 29/32",-3 29/32");	{Move pen}
LINETO(5 55/64",-3 29/32");	{Create a line}
ARROWHEAD(1);	{Set arrowhead}
MOVETO(3 29/32",-4 101/256");	{Move pen}
LINETO(5 55/64",-4 101/256");	{Create a line}
ARROWHEAD(11);	{Set arrowhead}
MOVETO(3 29/32",-4 113/128");	{Move pen}
LINETO(5 55/64",-4 113/128");	{Create a line}

Diagram 5-39

Statement Example 5-35 Output



Procedure ArrowSize(*arrowSize* : INTEGER)

Procedure ArrowSize sets the current arrow size setting within a MiniCad+ drawing. The parameter *arrowSize*, of type *Integer*, specifies the point size of the arrow.

Procedures ClosePoly and OpenPoly

When a ClosePoly statement is executed prior to Poly routines, the computer automatically encloses future polygons created. OpenPoly, on the other hand, allows created polygons to not be enclosed.

Statement Example 5-36 demonstrates these routines.

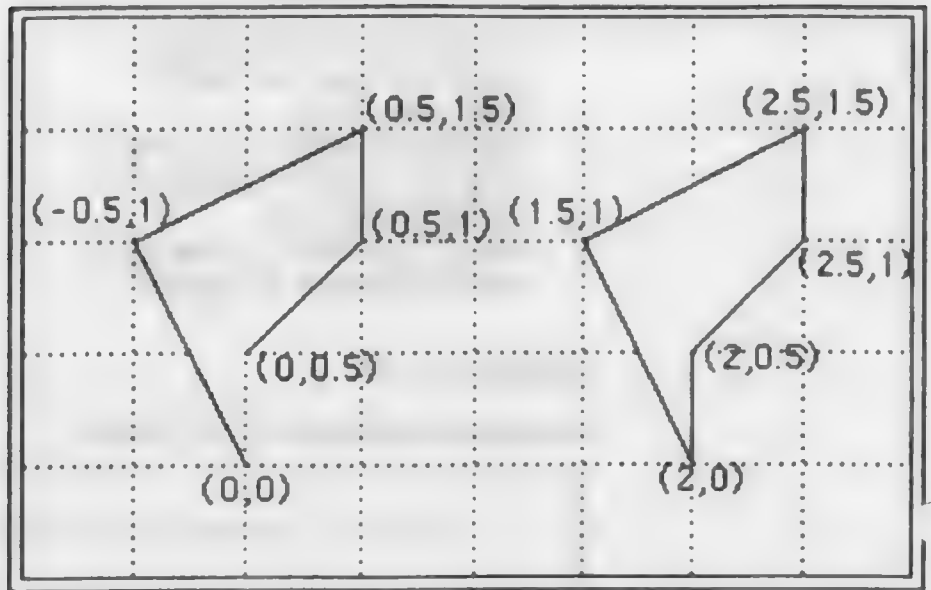
Statement Example 5-36

ClosePoly and OpenPoly Example

```
OPENPOLY;  
POLY(0,0,-0.5,1,0.5,1.5,0.5,1,0,0.5);  
CLOSEPOLY;  
POLY(2,0,1.5,1,2.5,1.5,2.5,1,2,0.5);
```


Diagram 5-40

OpenPoly and ClosePoly Examples



As Diagram 5-40 shows, the polygon created after the OpenPoly statement is not enclosed while the polygon created after the ClosePoly statement is automatically enclosed. The OpenPoly characteristic is the default setting.

Procedure CopyMode(Operation : INTEGER)

Procedure CopyMode sets the transfer mode of the active layer. The transfer mode setting determines the appearance of one layer in relation to another. There are four basic ways to set the appearance of the contents of a layer: Paint, Overlay, Invert and Erase. The Paint operation simply replaces the destination layer with the contents of the source layer. The Overlay operation superimposes the destination layer with the source layer. The Invert operation superimposes the destination layer with the source layer, however, it inverts the destination layer's black sections to white when they are covered by black sections of the source layer. Finally, operation Erase also superimposes the destination layer with the source layer, but it changes all sections in the destination layer to white if they are covered with a black section of the source layer. Each of these operations has a Not variant in which the black and white sections of the destination and source layers are inverted before the operation is performed. The parameter Operation, of type

integer, determines which operation is performed. Table 5-5 shows the possible values for the parameter *Operation* and Statement Example 5-37 shows an example of the procedure *CopyMode*.

Table 5-5

CopyMode Operations

<i>Operation</i>	<i>Value of Parameter Operation</i>
Paint	8
Overlay	9
Inverse	10
Erase	11
Not Paint	12
Not Overlay	13
Not Inverse	14
Not Erase	15

Statement Example 5-37

Procedure CopyMode Example

```
LAYER('First');      {Creates a layer entitled 'First'}
COPYMODE(11);        {Sets the transfer mode to Erase}
```

Procedure NameObject(ObjectName : STRING)

Procedure *NameObject* assigns the following created object a name. Its parameter, *ObjectName*, is of type *string*. Statement Example 5-38 demonstrates procedure *NameObject*.

Statement Example 5-38

Procedure NameObject Example

```
NAMEOBJECT('Tile');  {Create new name}
RECT(0,0.5,0.5,1);   {Assign name to the next created
                      object}
```

Procedure NameClass(ClassName : STRING)

Procedure *NameClass* assigns the following created objects a class. Its parameter, *ClassName*, is of type *string*. Statement Example 5-39 demonstrates procedure *NameClass*.

Statement Example 5-39

Procedure NameClass

```
NAMECLASS('SheetMetal');  
RECT(0,2,2,0);  
OVAL(-1,1,0,0);
```

Procedure FillPat(PatternNumber : INTEGER)

Procedure FillPat sets the pattern that will fill objects when they are created. The parameter PatternNumber may be any *integer* between 0 and 71. The pattern numbers correspond to the patterns shown in the Fill Menu (None equals 0, White equals one, etc.). Statement Example 5-40 demonstrates Procedure FillPat and Diagram 5-41 presents the available patterns.

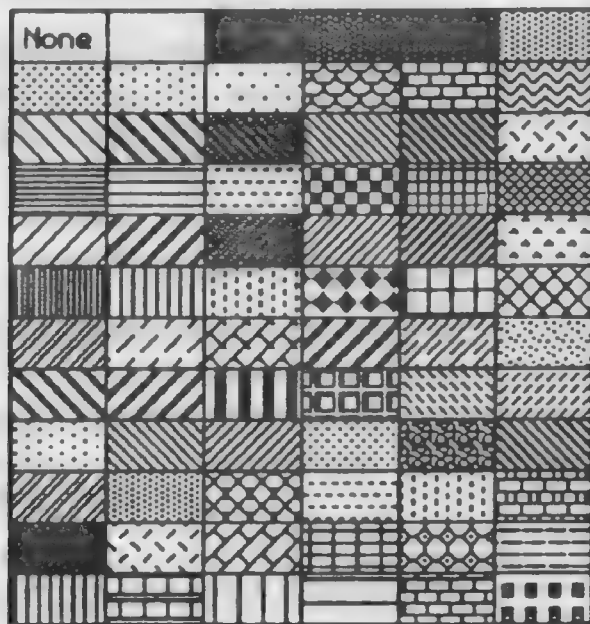
Statement Example 5-40

Procedure FillPat

```
FILLPAT(21);  
RECT(3 29/32",-4 77/512",5 315/512",-4 113/128");
```

Diagram 5-41

Fill Patterns



Procedure FillFore(Red,Green,Blue : LONGINT),
Procedure FillBack(Red,Green,Blue : LONGINT),
Procedure PenFore(Red,Green,Blue : LONGINT) and
Procedure PenBack(Red,Green,Blue : LONGINT)

Procedures FillFore, FillBack, PenFore and PenBack specify the appropriate current color settings. Their parameters, Red, Green and Blue are of type *longint* and are in the range [0..65535]. If the values of the parameters all equal 0, the color setting is black. If the parameters' values are all 65535, the color setting is white. Combinations of other values produce other color settings. Statement Example 5-41 demonstrates these routines.

Statement Example 5-41

Color Setting Example

```

FILLFORE(65535,65535,65535);
FILLBACK(0,0,0);
PENFORE(65535,65535,65535);
PENBACK(0,0,0);
RECT(3 107/256",-3 29/32",4 101/256",-4 101/256");

```










Procedures LckObjs and UnLckObjs

When an object is locked, it may be copied or duplicated, but it may not be changed in any other way. Procedure LckObjs locks all currently selected objects on the active layer. Conversely, Procedure UnLckObjs unlocks all selected locked objects on the active layer.

Procedure PenPat(PatternNumber : INTEGER)

Procedure PenPat sets the pen pattern (the line style). The parameter PatternNumber, of type *integer*, specifies the fill pattern or line style setting. If parameter PatternNumber is in the range [0..71], the corresponding fill pattern (as discussed with Procedure FillPat) is used as the pen pattern. If the parameter is in the range [-8..-1], a particular line style is set. The PatternNumber values for line styles are shown in Table 5-6.

Table 5-6

Line Styles	Value
	2
	-1
	-2
	-3
	-4
	-5
	-6
	-7
	-8

PatternNumber Values for Specific Line Styles

The default value of PatternNumber is 2, which represents the color black. Some patterns may not be noticeable unless the graphics pen is dimensioned to a particular size. Therefore, the Procedure PenSize may be needed. Procedure PenPat is demonstrated by Statement Example 5-42 and Diagram 5-42.

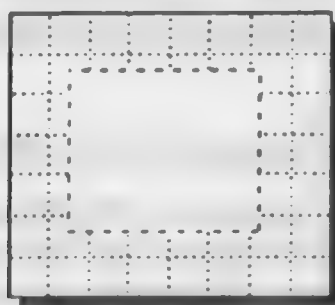
Statement Example 5-42

Procedure PenPat

```
PENPAT(25);  
RECT(3 803/1024",-3 389/512",4 477/512",-4 377/512");
```

Diagram 5-42

Output from Statement Example 5-42



Procedure PenSize(LineWeight : INTEGER)

Procedure PenSize specifies the line weight of the graphics pen. The parameter LineWeight, of type *integer*, specifies the line width in terms of mils. LineWeight's value is in the range [0..255]. With reference to pen size relative to pixel width, fourteen mils equal one pixel. Statement Example 5-43 and Diagram 5-43 demonstrate Procedure PenSize.

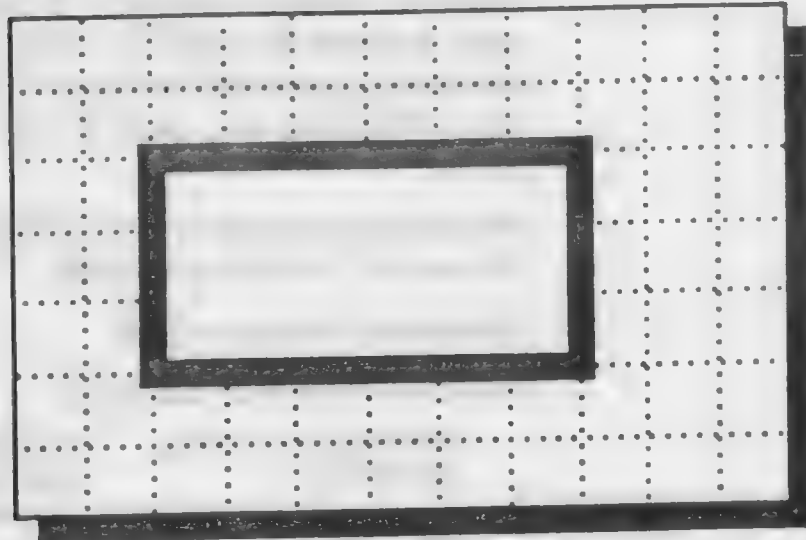
Statement Example 5-43

Procedure PenSize

```
PENSIZ(98);  
RECT(3 339/512",-3 29/32",5 65/512",-4 327/512");
```

Diagram 5-43

Statement Example 5-43 Output



Procedure Smooth(SmoothType : INTEGER)

Procedure Smooth is an option available for automatically smoothing polygons and freehand lines into constructions called spline curves. When Procedure Smooth is called, it smooths the selected objects on the active layer. Parameter SmoothType has three settings. These settings are shown in Table 5-7.

Table 5-7

Smoothing Options

<i>Smoothing Option</i>	<i>SmoothType</i>
No Smoothing	0
Bezier Spline	1
Cubic Spline	2

The Bezier Spline smoothing option is the familiar smoothing option that was available on former MiniCad releases. Its method of smoothing does not have the smoothed lines pass through the points of the polygon. Also, when attempting to reshape the polygon, a change to one part of a polygon is a local change (i.e. changing one part of the polygon does not affect other parts).

The Cubic Spline smoothing option has several differing features. When a polygon is smoothed, the smoothed lines do pass through the points of the polygon. Also, reshaping particular sections of a polygon are not local (i.e. reshaping one part of the polygon affects other parts).

Statement Example 5-44 and Diagram 5-44 demonstrate the use of Procedure Smooth.

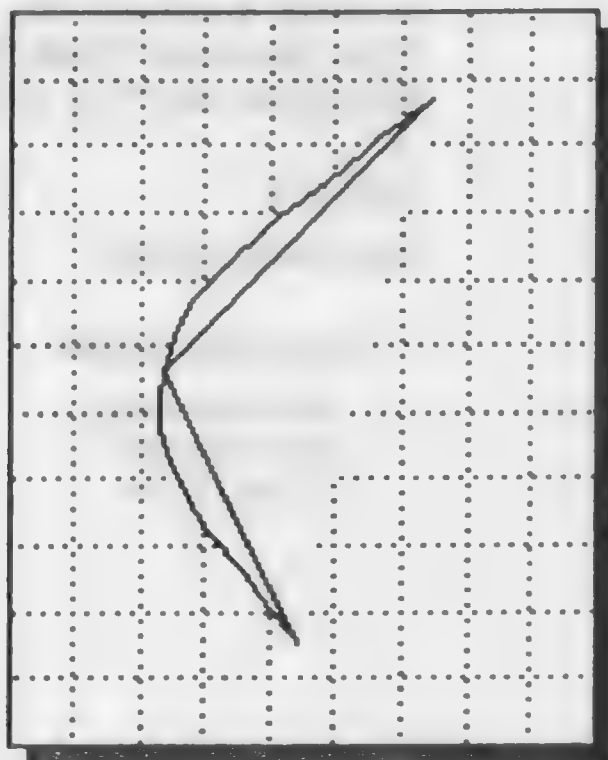
Statement Example 5-44

Procedure Smooth Example

POLY(0,0,-0.5,1,0.5,2);	{Create a standard polygon}
SMOOTH(2);	{Use the cubic spline smoothing option}
POLY(0,0,-0.5,1,0.5,2);	{Create a smoothed polygon}

Diagram 5-44

Smoothing Example



Procedures ShowLayer, HideLayer and GrayLayer

Procedures ShowLayer, HideLayer and GrayLayer specify the visibility of objects on a particular layer. Procedure ShowLayer displays the layer's contents in a normal visible manner. Procedure HideLayer does not display a layer's contents. Finally, procedure GrayLayer provides a gray appearance to a layer's contents. The default specification is constantly ShowLayer. Thus, if the programmer wishes to create two adjacent layers which are both grayed or hidden, he must specify this characteristic in both of them. Statement Example 5-45 demonstrates these routines.

Statement Example 5-45

Procedures ShowLayer, HideLayer and GrayLayer

LAYER('First');	{Create layer entitled 'First'}
GRAYLAYER;	{Gray the contents on the layer}
LAYER('Second');	{Create layer entitled 'Second'}
SHOWLAYER;	{Show the contents of the layer normally}
LAYER('Third');	{Create layer entitled 'Third'}
HIDELAYER;	{Do not display the layer's contents}
LAYER('Fourth');	{Create layer entitled 'Fourth'}
HIDELAYER;	{Do not display the layer's contents}

Procedure ShowClass(className : STRING)

Procedure HideClass(className : STRING)

Procedures ShowClass and HideClass change a class' visibility setting. The parameter className, of type *string*, specifies an existing class in the drawing to be changed. Procedure ShowClass sets a class' visibility to normal. Procedure HideClass sets a class' visibility to invisible.

Procedure TextFace([TextFaceType])

Procedure TextFace specifies the appearance of text. Text may be plain, bold, italic, underlined, outlined, or shadowed. It may also be a combination of several of these characteristics. In order to specify the text appear-

ance, the programmer provides the characteristics within brackets. Thus, if the programmer wishes to have text which is bold and italic, procedure TextFace would be written TextFace([Bold,Italic]). Likewise, if he wanted the text to be underlined, the programmer would write TextFace([Underline]). Finally, if the programmer wishes the text to be plain, he enters nothing between the brackets: TextFace([]). Statement Example 5-46 and Diagram 5-45 present procedure TextFace.

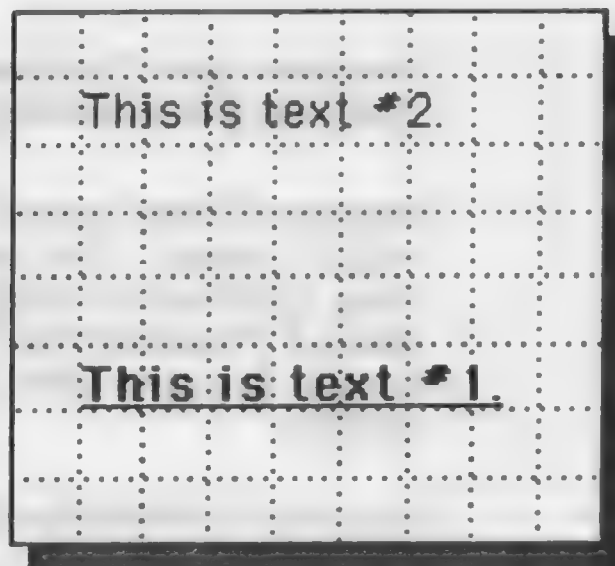
Statement Example 5-46

Procedure TextFace Example

```
TEXTFACE([Bold,Underline]);  
TEXTORIGIN(0,0);  
BEGINTEXT;  
    'This is text #1.'  
ENDTEXT;  
TEXTFACE([]);  
TEXTORIGIN(0,1);  
BEGINTEXT;  
    'This is text #2.'  
ENDTEXT;
```

Diagram 5-45

TextFace Example



Procedure TextFlip(FlipType : INTEGER)

Procedure TextFlip flips text vertically or horizontally. The parameter FlipType is of type *integer* and is in the range [0..2]. If FlipType equals 2, all text created after the procedure call will be flipped vertically. If FlipType equals 1, text will be flipped horizontally. Finally, if FlipType equals 0, no flipping occurs. Statement Example 5-47 and Diagram 5-46 present procedure TextFlip.

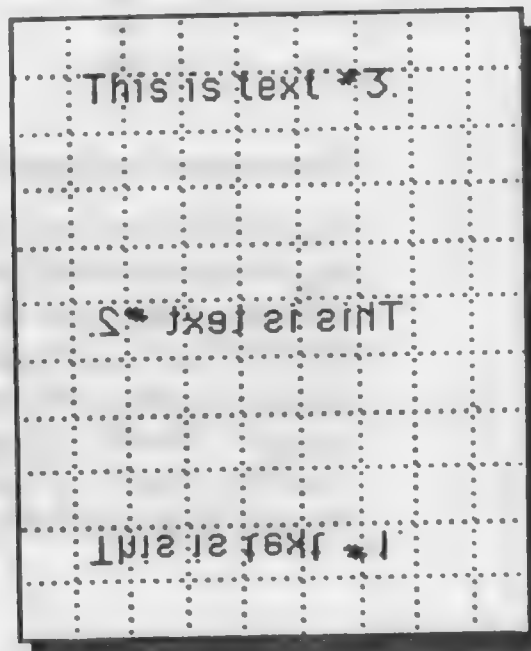
Statement Example 5-47

Procedure TextFlip Example

TEXTFLIP(2);	{Flip text vertically}
TEXTORIGIN(0,0);	{Create text}
BEGINTEXT;	
'This is text #1.'	
ENDTEXT;	
TEXTFLIP(1);	{Flip text horizontally}
TEXTORIGIN(0,1);	{Create text}
BEGINTEXT;	
'This is text #2.'	
ENDTEXT;	
TEXTFLIP(0);	{Do not flip text}
TEXTORIGIN(0,2);	{Create text}
BEGINTEXT;	
'This is text #3.'	
ENDTEXT;	

Diagram 5-46

Statement Example 5-47 Output



Procedure TextFont(FontNumber : INTEGER)

Procedure TextFont determines the type of font in which the text is displayed and printed. Font numbers range from 0 to 255. However, it is up to the user to manage the fonts available in the System File. The default font number is 3, which corresponds to the Geneva Font. Statement Example 5-48 exhibits Procedure TextFont. Table 5-8 provides several font styles and their corresponding numbers.

Statement Example 5-48

```
TEXTFONT(2);           {Sets text font to New York}
TEXTORIGIN(0,0);       {Create text}
BEGINTEXT;
    'This is the New York font.'
ENDTEXT;
```

Table 5-8

Text Fonts

<i>Font Number</i>	<i>Name</i>
0	System Font
1	Application Font
2	New York
3	Geneva
4	Monaco
5	Venice
6	London
7	Athens
8	San Francisco
9	Toronto
10	Seattle

Procedure TextJust(Justify : INTEGER)

Procedure TextJust justifies text which is created following the procedure call. The parameter Justify, which is of type *integer*, is in the range [1,2,3]. If Justify equals 1, created text is left justified relative to its text origin. If Justify equals 2, created text is centered relative to its text origin. Finally, if Justify equals 3, created text is right justified relative to its text origin. Statement Example 5-49 and Diagram 5-47 demonstrate this routine.

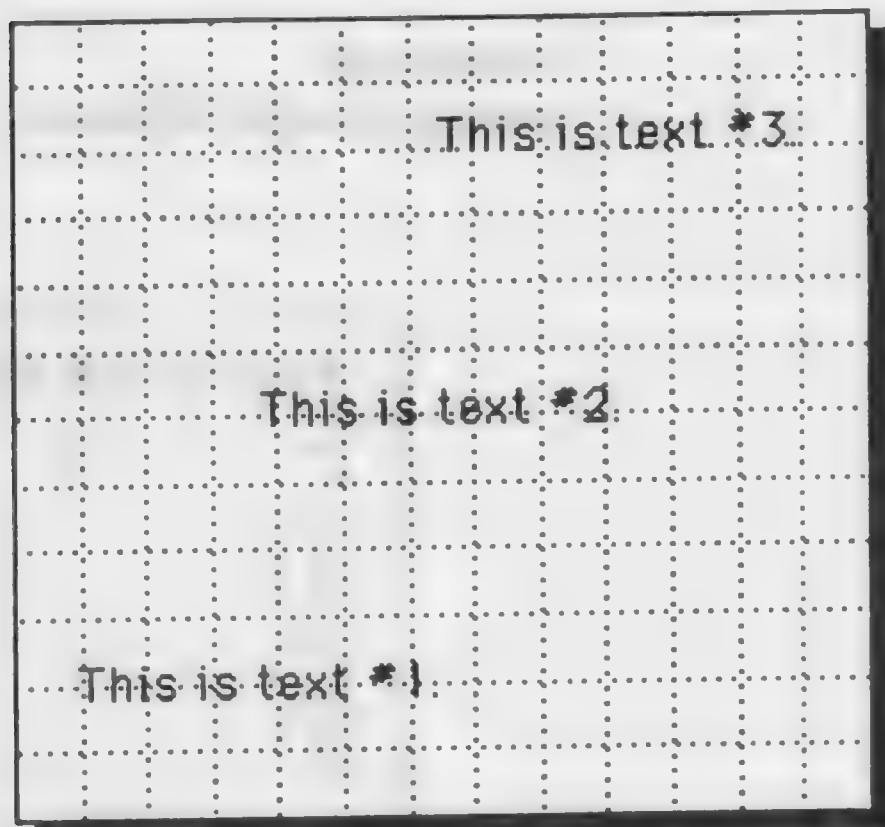
Statement Example 5-49

Procedure TextJust Example

TEXTJUST(3);	{Right justify text}
TEXTORIGIN(0,0);	{Create text}
BEGINTEXT;	
'This is text #1.'	
ENDTEXT;	
TEXTJUST(2);	{Center justify}
TEXTORIGIN(0,1);	{Create text}
BEGINTEXT;	
'This is text #2.'	
ENDTEXT;	
TEXTJUST(1);	{Left justify}
TEXTORIGIN(0,2);	{Create text}
BEGINTEXT;	
'This is text #3.'	
ENDTEXT;	

Diagram 5-47

Statement Example 5-49 Output



Procedure TextRotate(#Rotation : INTEGER)

Procedure TextRotate rotates text created after the procedure call. The parameter Rotation, of type *integer*, specifies the rotation angle. Statement Example 5-50 and Diagram 5-48 demonstrate this routine.

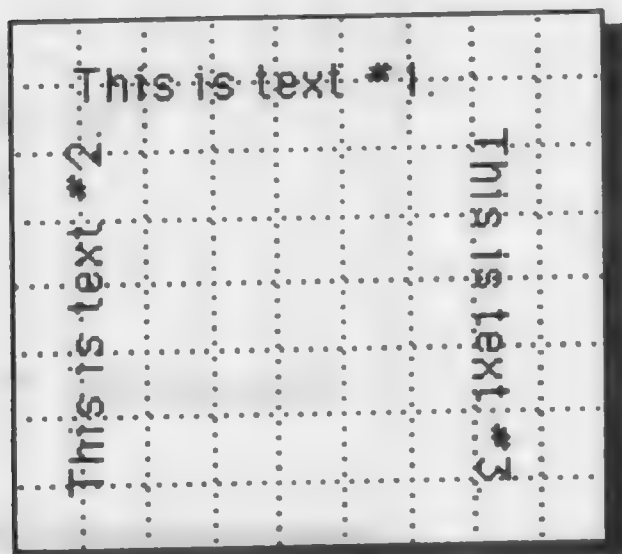
Statment Example 5-50

Procedure TextRotate Example

```
TEXTROTATE(#0);  
TEXTORIGIN(3 3669/4096",-4 1195/4096");  
BEGINTEXT;  
    'This is text #1.'  
ENDTEXT;  
TEXTROTATE(#90);  
TEXTORIGIN(3 843/1024",-4 71/128");  
BEGINTEXT;  
    'This is text #2.'  
ENDTEXT;  
TEXTROTATE(#270);  
TEXTORIGIN(5 699/2048",-4 151/256");  
BEGINTEXT;  
    'This is text #3.'  
ENDTEXT;
```

Diagram 5-48

Statement Example 5-50 Output



Procedure TextSize(Point : INTEGER)

Procedure TextSize determines the size of displayed text. The parameter point may be any *integer* value greater than or equal to 0, but the result will look best if the system folder has the font in that size. The next best result will occur if the given size is an even multiple of a size available for the font. If 0 is specified, the system font size (12 points) will be used. Statement Example 5-51 and Diagram 5-49 demonstrate this routine.

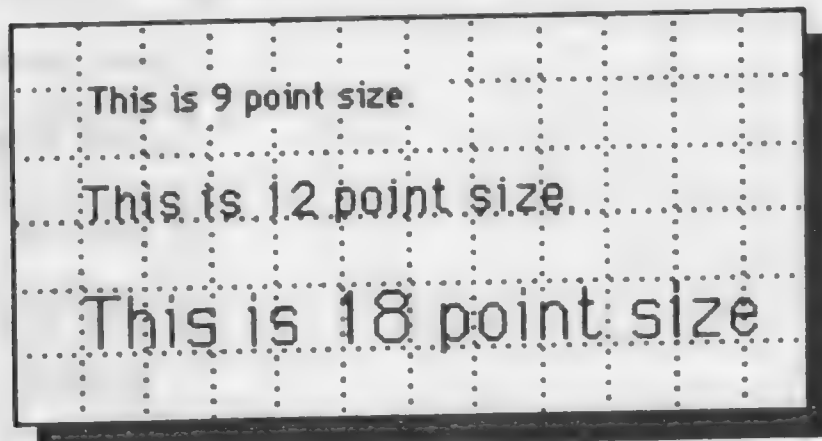
Statement Example 5-51

Procedure TextSize Example

```
TEXTSIZE(12);  
TEXTORIGIN(3 3783/4096",-3 3043/4096");  
BEGINTEXT;  
  'This is 12 point size.'  
ENDTEXT;  
TEXTSIZE(18);  
TEXTORIGIN(3 1863/2048",-4 519/4096");  
BEGINTEXT;  
  'This is 18 point size'  
ENDTEXT;  
TEXTSIZE(9);  
TEXTORIGIN(3 15/16",-3 3/8");  
BEGINTEXT;  
  'This is 9 point size.'  
ENDTEXT;
```

Diagram 5-49

Statement Example 5-51 Output



Procedure TextSpace(Spacing : INTEGER)

Procedure TextSpace specifies the spacing between text lines. The procedure only affects text created after its call. The parameter Spacing, of type *integer*, has three possible values: 2 (single spacing), 3 (1 1/2 spacing) and 4 (double spacing). Statement Example 5-52 and Diagram 5-50 present this routine.

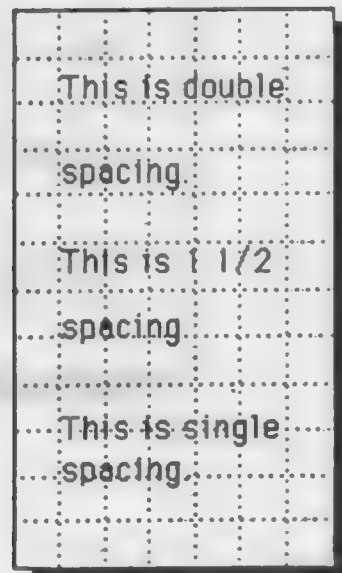
Statement Example 5-52

Procedure TextSpace Example

```
TEXTSPACE(4);  
TEXTORIGIN(3 2759/4096",-3 1479/2048");  
BEGINTEXT;  
  'This is double  
  spacing.'  
ENDTEXT;  
TEXTSPACE(3);  
TEXTORIGIN(3 2759/4096",-4 2605/4096");  
BEGINTEXT;  
  'This is 1 1/2  
  spacing.'  
ENDTEXT;  
TEXTSPACE(2);  
TEXTORIGIN(3 2759/4096",-5 1/2");  
BEGINTEXT;  
  'This is single  
  spacing.'  
ENDTEXT;
```

Diagram 5-50

Statement Example 5-52 Output



Section 5-4 Global Routines

Global procedures allow the user to determine the attributes or "characteristics" of tools in MiniCad+. These routines include: Procedure Absolute, Procedure Relative, Procedure AngleVar, Procedure NoAngleVar, Procedure DoubLine, Procedure DrwSize, Procedure GridLines, Procedure PenGrid, Procedure Redraw, Procedure SetOrigin, Procedure SetScale, Procedure Snap, Procedure SetUnits and Procedure Units.

Procedure Absolute

Procedure Absolute sets the point-designation method to absolute. When absolute is the current setting, all entered point designations are in absolute coordinates (e.g. the point designation (0,2) would be located at the graphic coordinate (0,2)). Statement Example 5-53 demonstrates this process while Diagram 5-51 shows the output.

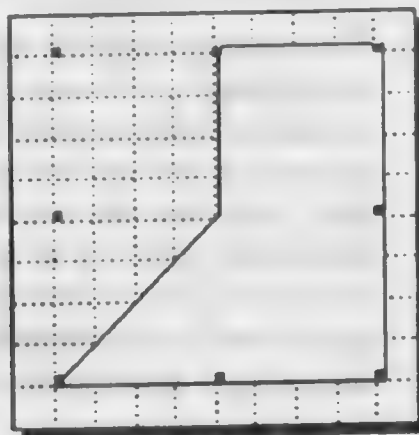
Statement Example 5-53

Procedure Absolute

ABSOLUTE;	{Set the computer in absolute mode}
CLOSEPOLY;	{Automatically close the polygon}
POLY(0,0,1,1,1,2,2,2,0);	{Create the polygon in absolute mode}

Diagram 5-51

Output from Statement Example 5-53



Procedure Relative

Procedure Relative sets the point-designation method to relative. Relative point designations move the graphics pen relative to its previous position (e.g. the point designation (0,2) would move the graphics pen two vertical units away from its present location). Statement Example 5-54 and Diagram 5-52 demonstrate Procedure Relative.

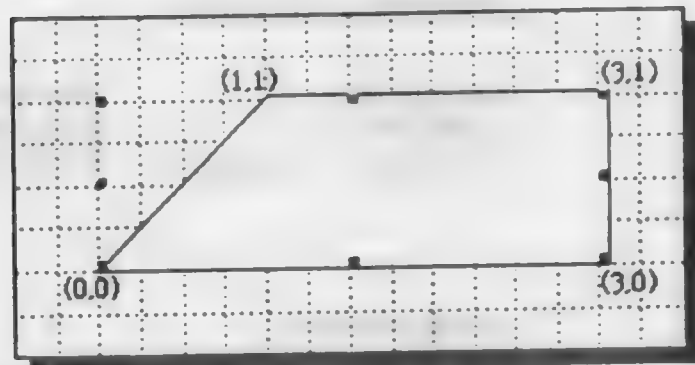
Statement Example 5-54

Procedure Relative

RELATIVE;	{Set the computer in relative mode}
MOVETO(0,0);	{Move the graphics pen to (0,0)}
CLOSEPOLY;	{Automatically close the polygon}
POLY(1,1,2,0,0,-1);	{Create the polygon in relative mode}

Diagram 5-52

Statement Example 5-54 Output



Procedures AngleVar and NoAngleVar

Procedures AngVar and NoAngVar are routines which are related to the use of variables for angle values. When writing a subprogram, there may be times when the user desires to use variables to represent angle values. Since the user has the capability of entering bearings (e.g. N,S,E,W), the use of a variable which begins with the letters N,S,E or W proposes a problem because the computer will not immediately know whether the user is entering a bearing or a variable. Thus, if the user calls

procedure AngleVar, this notifies the computer that variables will be used for angle values. Conversely, the procedure NoAngleVar informs the computer that no variables will be used for angle values and that any letters encountered should be treated as bearings. Since AngleVar and NoAngleVar are routines which inform the compiler about how to read angle values, every block within a subprogram must include an AngleVar call if applicable. Otherwise, the compiler assumes that all angle letters should be treated as bearings. Subprogram Example 5-3 demonstrates this process.

Subprogram Example 5-3

Procedures AngleVar and NoAngleVar Example

```

PROCEDURE AngVarDemo;
VAR
    AngValue : INTEGER;

    PROCEDURE Inner(v : REAL);
    BEGIN
        ANGLEVAR; {Note that inner subprogram needs call}
        LINETO(3",#v);
    END;

BEGIN
    AngValue := 65;
    ANGLEVAR;
    RELATIVE;
    LINETO(3",#AngValue);
    Inner(45);
    NOANGLEVAR;
    LINETO(3",#SE);
END;
RUN(AngVarDemo);

```

Procedure DoubLines(Distance : REAL)

Procedure DoubLines determines the distance used with the double line function. The parameter Distance, of type *real*, may be written in any of the coordinate distance formats discussed in Unit I. Statement Example 5-55 presents the command written in statement form. Diagram 5-53 demonstrates the routine's affect on a double line that is created within the same file.

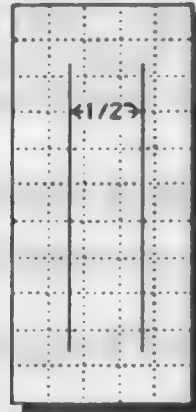
Statement Example 5-55

Procedure DoubLines Example

DOUBLINES(1/2"); {Set the double line distance}

Diagram 5-53

DoubLines Setting Example



Procedure DrwSize(Rows,Columns : INTEGER)

Procedure DrwSize determines the overall drawing sheet area of a MiniCad+ file. The parameters Rows and Columns, which are of type *integer*, represent the number of drawing sheets relative to the matrix found in the Drawing Size dialog box. Statement Example 5-56 shows the DrwSize routine in a statement form while Diagram 5-54 shows the Drawing Size dialog box with the new drawing size setting.

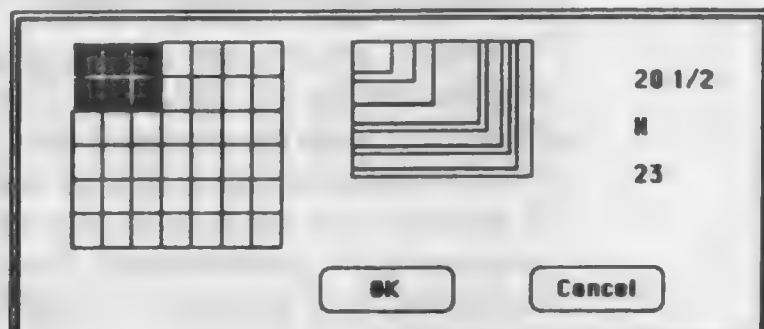
Statement Example 5-56

Procedure DrwSize Example

DrwSize(2,3); {Set drawing area}

Diagram 5-54

Drawing Size Dialog Box



Procedure GridLines(Distance : REAL)

Procedure GridLines sets the distance between the gridlines on the screen. The parameter Distance, of type *real*, may be written in any of the coordinate distance formats discussed in Unit I. Statement Example 5-57 exhibits the procedure in a statement form while Diagram 5-55 shows its result.

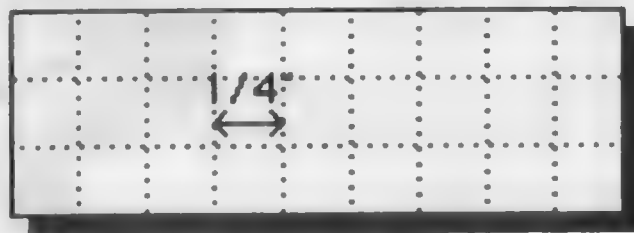
Statement Example 5-57

Procedure Gridlines Example

```
GRIDLINES(1/4");      {Set current gridline distance}
```

Diagram 5-55

Grid Line Distance



Procedure PenGrid(IncrementDistance : REAL)

Procedure PenGrid allows the user to determine the distance of increment movements for the graphics pen. The parameter IncrementDistance, of type *real*, may be written in any of the coordinate distance formats discussed in Unit I. Statement Example 5-58 presents this routine in statement form.

Statement Example 5-58

Procedure PenGrid Example

```
PENGRID(1/8");      {Set current pengrid increment  
distance}
```

Procedure ReDraw

When writing subprograms in MiniPascal, there may be times when the programmer deems it necessary to refresh the screen prior to the default redraw at the termination of his program. Procedure ReDraw performs this operation. Subprogram Example 5-4 shows this routine within a subprogram.

Subprogram Example 5-4

Procedure Redraw

```
PROCEDURE RedrawEx;
VAR
    count : INTEGER;
BEGIN
    ANGLEVAR;
    BEGINXTRD(0,1);
    RECT(0,1,1,0);
    ENDXTRD;
    DSELECTALL;
    FOR count := 1 TO 100 DO
    BEGIN
        ROTATE3D(#0,# count * 0.3,#0);
        REDRAW;
    END;
END;
RUN(RedrawEx);
```

Procedure SetOrigin(X,Y : REAL)

The user may wish to redefine the default location of the origin in order to renumber the coordinate system in MiniCad+ file. Procedure SetOrigin performs this operation. The parameters X and Y, which are of type *real*, may be written in any of the coordinate distance formats discussed in Unit I. When procedure SetOrigin is called, these values designate the new numerical origin for the drawing. Statement Example 5-59 demonstrates this routine.

Statement Example 5-59

Procedure SetOrigin

```
SetOrigin(2,2);           {Sets new origin to (2,2)}
```


Procedure SetScale(ActualSize : REAL)

Procedure SetScale adjusts the paper scale. The paper scale is the relationship of an object's printed size to its actual size. This relationship is a ratio which is expressed as PRINTED SIZE : ACTUAL SIZE, where printed size always has a value of 1. The parameter ActualSize is of type *real*. Statement Example 5-60 shows Procedure SetScale as a statement.

Statement Example 5-60

```
SETSCALE(2);    {Prints and displays objects 1/2 their actual size}
```

Therefore, an object which is actually 2" x 2" (See Diagram 5-56) will be printed and displayed on the screen 1/2 its actual size (See Diagram 5-57).

Diagram 5-56

Actual Size of an Object

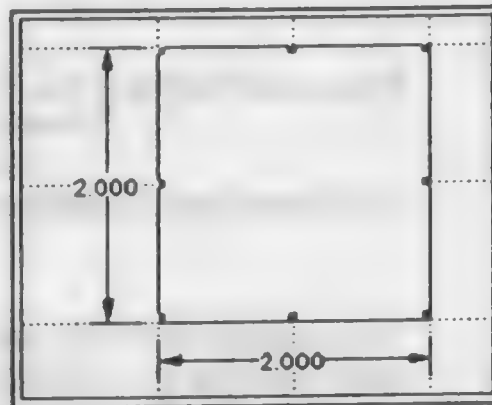
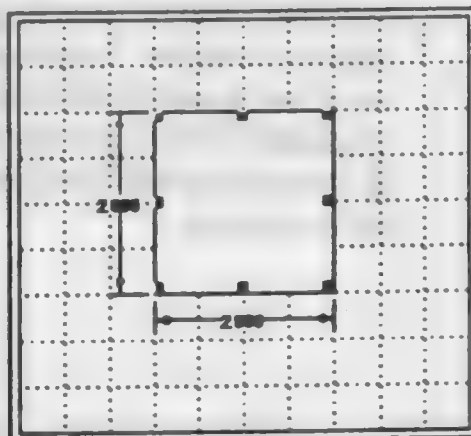


Diagram 5-57

Printed Size of an Object



Procedure Snap(Grid, Object, Locus : BOOLEAN)

Procedure Snap activates the 'Snap' options. There are three snap options available, these include: Snap to Grid, Snap to Object, and Snap to Locus. Each parameter, which is of type *boolean*, may have only one of two values: true or false. The setting of each parameter determines the configuration of Procedure Snap. For example, if the user desires to have the Snap to Grid operation active and the other two operations inactive, Procedure Snap would look like the following: Snap(TRUE, FALSE, FALSE). Likewise, if the user wishes to have the Snap to Object and Snap to Locus operations active and the Snap to Grid operation inactive, Procedure Snap would be stated Snap(FALSE, TRUE, TRUE). Statement Example 5-61 demonstrates Procedure Snap.

Statement Example 5-61

Procedure Snap

```
SNAP(TRUE, FALSE, TRUE);    {activates snap to grid and  
                             snap to locus}
```

```
Procedure SetUnits( Fraction, DisplayAccuracy : LONGINT;  
                  Format : INTEGER; UnitsPerInch : REAL;  
                  UnitMark, SqrUnitMark : STRING)  
and Procedure Units(StandardUnit : INTEGER;  
                  DisplayAccuracy : LONGINT)
```

Procedures SetUnits and Units determine the current unit settings in a MiniCad+ file. Procedure SetUnits provides the capability of entering custom units while Procedure Units allows the user to select one of the standard unit settings.

Procedure SetUnits accepts six parameters. The parameter Fraction, of type *longint*, specifies the smallest possible unit value. For example, if Fraction equals 4096, the smallest possible unit value would be 1/4096. Thus, the parameter Fraction provides a minimum accuracy unit. If the parameter Fraction equals zero, a 'Division by zero' error occurs.

The parameter Display Accuracy, of type *longint*, sets the display accuracy value. This value provides the minimum display value. If fractional units are used (this setting is determined by the parameter Format, which is discussed shortly) this value represents the minimum fractional value displayed. For instance, if Display Accuracy equals 16, the minimum value displayed in the coordinate boxes is 1/16. If decimal units are used, it represents the display accuracy of the decimal portion of a value. Thus, if Display Accuracy equals 1000, a value will be displayed as 3.000. The parameter Display Accuracy may equal zero if decimal units are used and the programmer does not wish to display the decimal portion of a number. However, Display Accuracy may not equal zero if fractional units are used, doing so will produce a 'Divide by Zero' error.

Parameter Format, of type *integer*, determines whether unit values are displayed in a decimal format, a fractional format, a decimal feet and inches format or a fractional feet and inches format. Format has four possible values. Table 5-9 presents these values and the corresponding displayed unit forms.

Table 5-9

Possible Format Values and Unit Formats

<i>Format Value</i>	<i>Unit Format</i>
0	Decimal
1	Fractional
2	Decimal Feet and Inches
3	Fractional Feet and Inches

The parameter UnitsPerInch, of type *real*, specifies the number of units that will equal an inch. Thus, if the programmer wishes to create a new unit type in which two units equal one inch, the value of UnitsPerInch would be two.

The parameter UnitMark, of type *string*, specifies the new unit's character representation after numeric values (such as 'm' represents meters). Therefore, if the UnitMark equals 'dc' and a numeric value equals 10, it

will be displayed as 10 dc. Likewise, the parameter SqrUnitMark, also of type *string*, represents the new unit in a squared form. Neither UnitMark nor SqrUnitMark can be greater than eight characters.

Procedure Units accepts two parameters, StandardUnit and DisplayAccuracy. The parameter StandardUnit is of type *integer* and the parameter DisplayAccuracy is of type *longint*. The parameter StandardUnit is in the range [1..6], each value representing a different standard unit setting (these settings are displayed in Table 5-10). The parameter DisplayAccuracy determines the display accuracy setting as described with Procedure SetUnits. The value of DisplayAccuracy is limited to the available accuracy settings which are shown in the Unit dialog box. Therefore, if the standard unit setting is in a fractional format, the possible values of DisplayAccuracy are [2,4,8,16,32,64]. However, if the standard unit setting is in a decimal format, the possible values of DisplayAccuracy are [0, 10, 100, 1000, 10000, 100000, 1000000, 10000000, 1000000000].

Table 5-10

Standard Unit Settings

Standard Unit Value	Fraction	Format	Units Per Inch	Unit Mark	Square Unit Mark
1	4096	3	1.0	"	sqr ft
2	1000	0	25.4	mm	mm^2
3	1000	0	2.54	cm	cm^2
4	1000	0	1.0	"	in^2
5	1000	0	0.0254	m	m^2
6	4096	2	1.0	"	sqr ft

Statement Example 5-62 demonstrates Procedure SetUnits and Procedure Units.

Statement Example 5-62

Procedures SetUnits and Units Example

```
SETUNITS(1000,1000,0,2,'dc','dc^2'); {Create new units}
UNITS(3,100);                        {Set units to standard-
                                     centimeters}
```

Section 5-5 Alteration Routines

Alteration procedures allow the user to manipulate objects which are currently displayed in a Minicad+ file. These procedures include: Procedure DeleteObjs, Procedure Duplicate, Procedure FlipHor, Procedure FlipVer, Procedure Forward, Procedure Backward, Procedure MoveFront, Procedure MoveBack, Procedure MoveObjs, Procedure Move3D, Procedure Rotate, Procedure Rotate3D, Procedure Scale, Procedure DelClass, Procedure DelName, Procedure VSave, Procedure VRestore and Procedure VDelete.

Procedure DeleteObjs

Procedure DeleteObjs deletes all selected objects on the active layer. Statement Example 5-63 demonstrates this routine.

Statement Example 5-63

Procedure DeleteObjs Example

```
DELETEOBS;           {Delete all selected objects}
```

Procedure Duplicate(X,Y : REAL)

Procedure Duplicate makes a copy of the currently selected object or objects on the active layer and moves the copy X coordinates horizontally and Y coordinates vertically. The coordinate movement of copied objects is relative to the original objects present location. Parameters X and Y, of type *real*, may be written in any applicable coordinate format. Statement Example 5-64 and Diagram 5-58 demonstrate this routine.

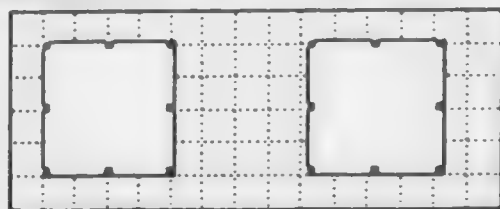
Statement Example 5-64

Procedure Duplicate Example

```
RECT(0,1,1,0);       {Create a rectangle}
DUPLICATE(2,0);       {Duplicate the rectangle and move
                      the copy two units to the right}
```

Diagram 5-58

Statement Example 5-64 Output



Procedures FlipHor and FlipVer

Procedures FlipHor and FlipVer allow the user to horizontally or vertically mirror a graphic object in a Minicad+ file. These routines produce the same affects as the menu items "Flip Horizontal" and "Flip Vertical." Statement Example 5-65 and Diagram 5-59 demonstrate Procedure FlipVer.

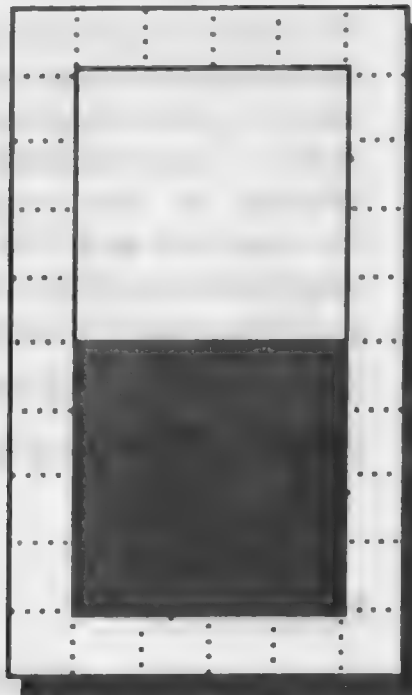
Statement Example 5-65

Procedure FlipVer Example

BEGINGROUP;	{Start creating a group}
FILLPAT(2);	{Set fill pattern to black}
RECT(0,1,1,0);	{Create a black-filled rectangle}
FILLPAT(1);	{Set fill pattern to white}
RECT(0,0,1,-1);	{Create a white-filled rectangle}
	{At this point, the black rectangle is above the white rectangle}
ENDGROUP;	{Finish creating a group}
FLIPVER;	{Flip the group vertically}

Diagram 5-59

Statement Example 5-65 Output



Procedures **Forward**, **Backward**, **MoveFront** and **MoveBack** Objects in a MiniCad+ file are drawn in the order that they are created on a layer. This ordering of objects creates a list in which the first created object is at the back of the list and all proceeding objects are placed in the list one position higher than the previously created object. This ordering allows the most currently created objects to appear in front of previously created objects. If some objects are filled with a particular pattern, they may obscure the user's view of previously created objects. Thus, the user may wish to "move" the list position of some objects.

Procedures **Forward** and **Backward** move currently selected objects one position in the list. Therefore, if a white-filled object obscures a black-filled object (because the white-filled object is one position higher in the list than the black-filled object) and the black-filled object is selected, the procedure call **MoveForward** will place the black-filled object on top of the white-filled object (the black-filled object is moved one position higher in the list). If several objects are selected when either procedure **Forward** or **Backward** is called, all selected objects move one position in the list.

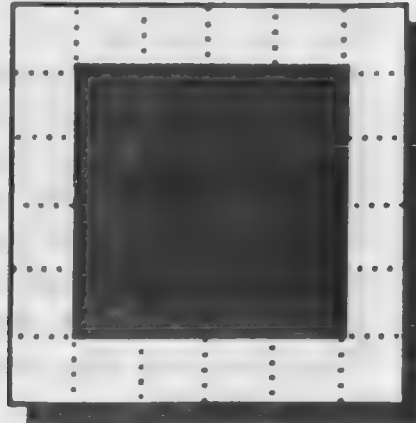
Procedures **MoveFront** and **MoveBack** move the position of objects in the graphics list also, however, they move the currently selected object(s) to the very front of the list (procedure **MoveFront**) or to the very back of the list (procedure **MoveBack**). If more than one object is selected, the selected objects maintain their relative positions after the move. Statement Example 5-66 and Diagram 5-60 demonstrate procedure **MoveBack**.

Statement Example 5-66

Procedure **MoveBack** Example

OVAL(0,1,1,0);	{Create an oval}
FILLPAT(2);	{Change fill pattern to black}
RECT(0,1,1,0);	{Create a rectangle}
FILLPAT(4);	{Change fill pattern to grey}
DSELECTALL;	{Deselect all objects}
RRECT(0,1,1,0,0.5,0.5);	{Create a rounded rectangle}
	{At this point the grey rounded rectangle is at the top of the list and is covering the other two objects}
MOVEBACK;	{Move rounded rectangle to back of list}
	{The next object in the list is the rectangle}

Diagram 5-60
Statement Example 5-66 Output



**Procedure MoveObjs(dX,dY : REAL;
AllLayers, AllObjects : BOOLEAN)**

Procedure MoveObjs moves created object(s) dX coordinates horizontally and dY coordinates vertically. The coordinate movement of objects is relative to their present location. Parameters dX and dY, of type *real*, may be written in any applicable coordinate format. The parameters AllLayers and AllObjects, of type *boolean*, specify which objects should be moved. Table 5-11 presents the possible values of the parameters AllLayers and AllObjects and their affect upon the movement of objects. Statement Example 5-67 demonstrates this routine.

TABLE 5-11

The Values of AllLayers and AllObjects

<i>AllLayers Value</i>	<i>AllObjects Value</i>	<i>Move Objects Only On Active Layer</i>	<i>Move Only Selected Objects</i>
TRUE	TRUE	NO	NO
TRUE	FALSE	NO	YES
FALSE	TRUE	YES	NO
FALSE	FALSE	YES	YES

Statement Example 5-67

Procedure MoveObjs Example

```
RECT(0,1,1,0);           [Create a rectangle]
RRECT(1,1,2,0,0.5,0.5);  [Create a rounded-
                           rectangle]
MOVEOBS(3,0,FALSE,FALSE); [Move both objects 3 units-
                           to the right]
```

Procedure Move3D(XDist,YDist,ZDist : REAL)

Procedure Move3D moves the most recently created three-dimensional object the distance designated by the parameters XDist, YDist and ZDist, all of type *real*. The parameters may be written in any of the distance formats described earlier in the manual. The object is moved relative to its center. If the programmer is viewing the screen from the Top View, then XDist moves the object to the left or right, YDist moves the object up or down and ZDist moves the object toward or away from the viewer. Statement Example 5-68 demonstrates Procedure Move3D and Diagram 5-61 shows its output.

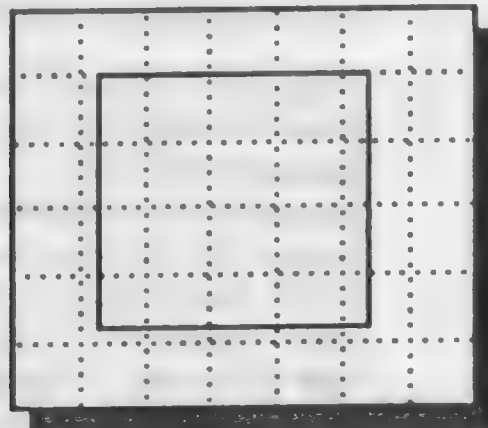
Statement Example 5-68

Procedure Move3D

```
BeginXtrd(0°,1 1/1024°);
  Rect(549/1024°,2 169/256°,1 275/512°,1 375/512°);
EndXtrd;
Move3D(1°,1°,1°);
```

Diagram 5-61

Output from Statement Example 5-68



Procedure Rotate(#DegreeValue : REAL)

Procedure Rotate rotates the currently selected objects on the active layer. The parameter DegreeValue, of type *real*, represents a degree value and may be written in any of the degree formats discussed in Unit I. Clockwise rotations are given a negative sign while counter-clockwise rotations have a positive sign. Note the pound sign (#) which is required of all angle values. Also, if DegreeValue is a variable, the procedure AngleVar will need to be called prior to the statement. Statement Example 5-69 and Diagram 5-62 demonstrate this routine.

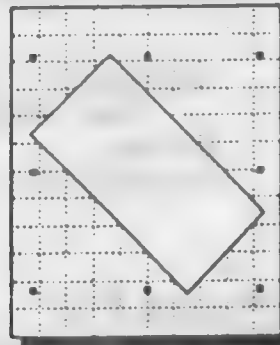
Statement Example 5-69

Procedure Rotate Example

RECT(0,2,1,0);	{Create a rectangle}
ROTATE(#45);	{Rotate the object 45°}

Diagram 5-62

Statement Example 5-69 Output



Procedure Rotate3D(#XAngle,#YAngle,#ZAngle : REAL)

Procedure Rotate3D rotates the most recently created three-dimensional object by the values specified in the parameters XAngle, YAngle and ZAngle, all of type *real*. An angle value (other than 0°) in the XAngle parameter rotates the 3D object around its X axis. An angle value (other than 0°) in the YAngle parameter rotates the 3D object around its Y axis. Finally, an angle value (other than 0°) in the ZAngle parameter rotates the 3D object around its Z axis. Statement Example 5-70 demonstrates Procedure Rotate3D and Diagram 5-63 shows its output.

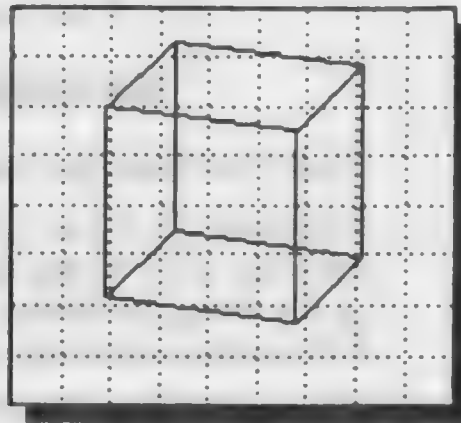
Statement Example 5-70

Procedure Rotate3D

```
BeginXtrd(-2133/4096°,1967/4096°);  
  Rect(1657/4096°,1 519/2048°,1 1661/4096°,517/2048°);  
EndXtrd;  
Rotate3D(#21° 10' 22",#-18° 44' 50",#-7° 5' 45");
```

Diagram 5-63

Output from Statement Example 5-70



Procedure Scale(ScaleX,ScaleY : REAL)

Procedure Scale allows the user to rescale a selected object on the active layer. The parameters ScaleX and ScaleY, of type *real*, determine the precise scaling factor. Values for ScaleX and ScaleY which are less than 1 will reduce the object, while values greater than 1 will enlarge it (values of zero have no effect). Negative values reverse as well as rescale a selection. The selection, whether one object, a group, or several objects, will always rescale around its center (i.e. its center will remain stationary to the rest of the drawing. Statement Example 5-71 and Diagram 5-64 demonstrate Procedure Scale.

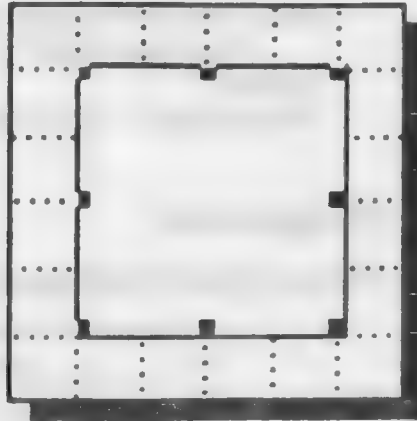
Statement Example 5-71

Procedure Scale

```
RECT(0,0.5,0.5,0);    {creates a rectangle}  
SETSCALE(2,2);         {rescales rectangle to twice its size}
```

Diagram 5-64

Statement Example 5-71 Output



Procedure DelClass(className : STRING)

Procedure DelClass deletes a class from a MiniCad+ drawing. The routine deletes the class which is specified by the *string* parameter className. Statement Example 5-72 demonstrates this routine.

Statement Example 5-72

```
DelClass('MyClass');           {Delete class name}
```

Procedure DelName(name : STRING)

Procedure DelName deletes an object name from a MiniCad+ drawing. The routine deletes the object name which is specified by the *string* parameter name. Statement Example 5-73 demonstrates this routine.

Statement Example 5-73

```
DelName('MyName');            {Delete object name}
```

Procedure VSave(viewName : STRING)

Procedure VSave saves the current view and stores it under the name specified by the *string* parameter viewName.

Procedure VRestore(viewName : STRING)

Procedure VRestore restores the saved view which is specified by the *string* parameter viewName.

Procedure VDelete(viewName : STRING)

Procedure VDelete deletes the saved view which is specified by the *string* parameter viewName.

Section 5-6 Utility Routines

Utility routines are routines which perform basic operations for the programmer. These routines include: Procedure SysBeep, Function FndError, Function Date and Procedure Wait.

Procedure SysBeep

Procedure SysBeep creates a beep sound when called. This procedure may be used in several instances including debugging and alerting the user.

Function FndError: BOOLEAN;

Function FndError (FoundError) returns whether an error has occurred within a MiniPascal subroutine. This function will primarily aid programmers in debugging runtime errors and trapping invalid values passed to pre-defined routines. The function is updated after each statement, thus if an early line generates an error, the function will not continue to be true for future statement lines. For example:

```
PROCEDURE Test;
VAR
  int1,int2 : INTEGER;
  s : STRING;
BEGIN
  int1 := IntDialog('Enter an integer:','0');
  IF FndError THEN AlrtDialog('Test #1');
  int2 := IntDialog('Enter an integer:','0');
  IF FndError THEN AlrtDialog('Test #2');
  s := Date(int1,int2);
  IF FndError THEN AlrtDialog('Test #3');
  writeln('s = ',s);
END;
RUN(Test);
```

Function Date(dateFormat, infoFormat : INTEGER) : STRING;

The Function Date returns a string which contains date and time information. The programmer can specify the format of the information through the parameters dateFormat and infoFormat.

The parameter *infoFormat* determines whether the string contains both the date and time, just the date or just the time. The possible values of *infoFormat* are shown below:

<i>Value of infoFormat</i>	<i>Setting</i>
0	Return the date and time.
1	Return the date.
2	Return the time.

The parameter *dateFormat* determines the format of date information. (Note: this parameter does not affect the string if only time information is being returned). The possible values of *dateFormat* are shown below:

<i>Value of dateFormat</i>	<i>Date Information Format</i>
0	Full form.
1	Abbreviated form.
2	Short form.

Assuming the date is Friday, November 18, 1988 and the time is 10:42:24 AM, the function call: *Date*(0,0) would return:

Friday, November 18, 1988 10:42:24 AM

The function call: *Date*(0,1) would return

Friday, November 18, 1988

The function call: *Date*(0,2) would return

10:42:24 AM

The function call: *Date*(1,1) would return

Fri, Nov 18, 1988

The function call: *Date*(2,0) would return

11/18/88 10:42:24 AM

Procedure *Wait*(seconds : INTEGER);

Procedure *Wait* continues to execute until the number of seconds specified in the parameter *seconds* has passed.

Section 5-7 Dialog Routines

Dialog routines display dialog boxes which allow the programmer to request information from the user. These routines include: Function AngDialog, Function DistDialog, Function IntDialog, Procedure PtDialog, Function RealDialog, Function StrDialog, Function YNDialog, Procedure AlrtDialog, Procedure AngDialog3D, Procedure PtDialog3D and Function DidCancel.

Function AngDialog(request, default : STRING) : REAL

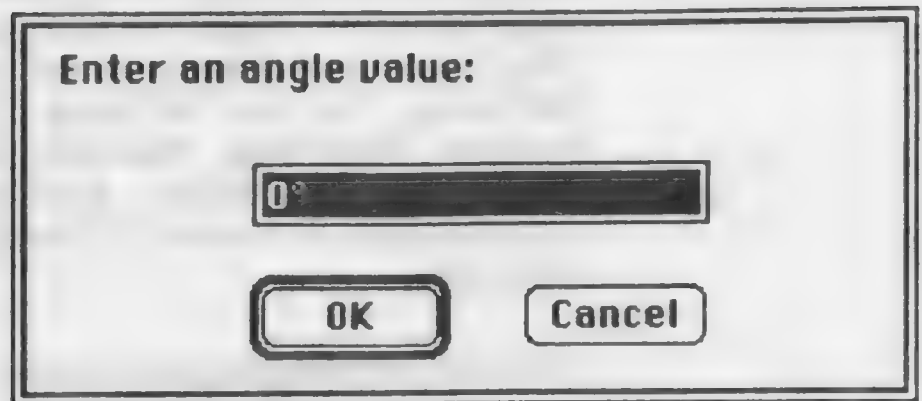
Function AngDialog, which returns a value of type *real*, displays a dialog box which requests the user to enter an angle value. The parameters RequestStr and Default, of type *string*, allow the programmer to specify the string which requests the information from the user and the default angle value which appears in the angle value display box. The angle dialog box will accept angle values in any of the angle formats discussed in Unit I. If the user enters an invalid angle value, the dialog box will produce a beep and display the initial default value. Also, if the user selects cancel from the dialog box, the function will return the default value. Statement Example 5-74 demonstrates function AngDialog in a sub-program. Diagram 5-65 presents the dialog box which is displayed during the execution of Statement Example 5-74.

Statement Example 5-74

```
PROCEDURE AngleDialog;  
VAR  
    AngleValue : REAL;  
BEGIN  
    AngleValue := ANGDILOG('Enter an angle value:', '0°');  
    RECT(0,1,1,0);  
    ANGLEVAR;  
    ROTATE(#AngleValue);  
END;  
RUN(AngleDialog);
```

Diagram 5-65

Dialog Box from Statement Example 5-74



Function DistDialog(RequestStr,Default : STRING) : REAL

Function DistDialog, which returns a value of type *real*, displays a dialog box which requests the user to enter a distance value. The parameters RequestStr and Default, of type *string*, allow the programmer to specify the string which requests the information from the user and the default distance value which appears in the distance value display box. The distance dialog box will accept distance values in any of the distance formats discussed in Unit I. If the user enters an invalid distance value, the dialog box will produce a beep and re-display the initial default value. Also, if the user selects cancel from the dialog box, the function will return the default value. Statement Example 5-75 demonstrates function DistDialog in a subprogram. Diagram 5-66 presents the dialog box which is displayed during the execution of Statement Example 5-75.

Statement Example 5-75

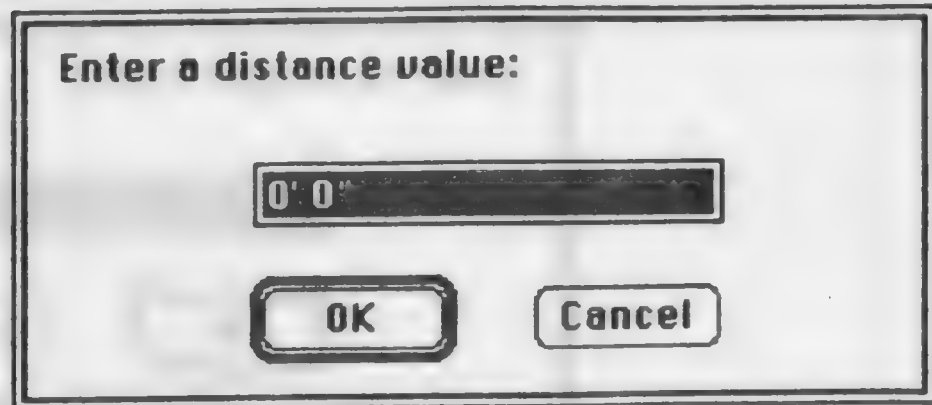
Function DistDialog Example

```
PROCEDURE DistanceDialog;           {Create subprogram}
VAR
    DistValue : REAL;                {Declare real variable}
BEGIN
    DistValue := DISTDIALOG("Enter a distance value:", "0" "0");
                                   {Get distance value}
    RECT(DistValue,#90,DistValue,#0);
                                   {Create a rectangle using the distance-angle-coordinate method}

END;
RUN(DistanceDialog);
```

Diagram 5-66

Dialog Box from Statement Example 5-75



**Function IntDialog(RequestStr,Default : STRING) :
INTEGER**

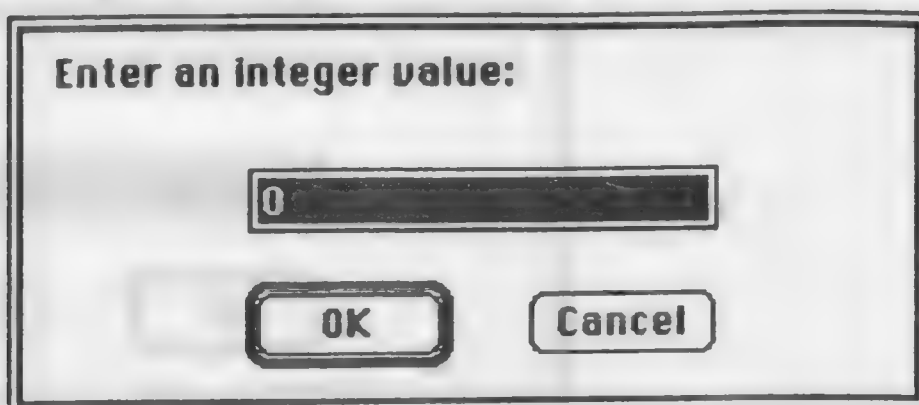
Function IntDialog, which returns a value of type *integer*, displays a dialog box which requests the user to enter an *integer* value. The parameters RequestStr and Default, of type *string*, allow the programmer to specify the string which requests the information from the user and the default *integer* value which appears in the *integer* value display box. The *integer* dialog box will accept *integer* values written in any of the distance formats discussed in Unit I. If the user enters an invalid value, the dialog box will produce a beep and re-display the initial default value. Also, if the user selects cancel from the dialog box, the function will return the default value. Statement Example 5-76 demonstrates function IntDialog in a subprogram. Diagram 5-67 presents the dialog box which is displayed during the execution of Statement Example 5-76.

Statement Example 5-76

Function IntDialog Example

```
PROCEDURE IntegerDialog;      {Create subprogram}
VAR
  IntValue : INTEGER;         {Declare integer variable}
BEGIN
  IntValue := INTDIALOG(Enter an integer value:', 0');
                                {Get integer value}
  WRITELN('The user entered: ',IntValue);
                                {Display integer value}
END;
RUN(IntegerDialog);
```

Dialog Box from Statement Example 5-76



```

Procedure PtDialog(RequestStr, DefaultX,
                    DefaultY : STRING; X,Y : REAL)

```

Procedure PtDialog displays a dialog box which requests the user to enter a coordinate (point) value. The parameters RequestStr, DefaultX and DefaultY, of type *string*, allow the programmer to specify the string which requests the information from the user and the default x and y values which will appear in the point value display box. The parameters X and Y, of type *real*, return the numeric coordinate values entered by the user. The point dialog box will accept coordinate values written in any of the coordinate formats discussed in Unit I. If the user enters an invalid coordinate value, the dialog box will produce a beep and re-display the initial default value. Also, if the user selects cancel from the dialog box, the function will return the default coordinate values. Statement Example 5-77 demonstrates Procedure PtDialog in a subprogram. Diagram 5-68 presents the dialog box which is displayed during the execution of Statement Example 5-77.

Statement Example 5-77

Procedure PtDialog Example

```

PROCEDURE PointDialog;           {Create subprogram}
VAR
    X,Y : real;                 {Declare real variables}
BEGIN
    PTDIALOG('Enter a coordinate.','0','0',X,Y);
                                {Get coordinate values}
    WRITELN('X = ',X:5:2,' Y = ',Y:5:2);
                                {Display coordinate values}
END;
RUN(PointDialog);

```

Diagram 5-68

Dialog Box from Statement Example 5-77

Enter a coordinate:

H: 0

Y: 0

OK Cancel

Function RealDialog(RequestStr,Default : STRING) : REAL

Function RealDialog, which returns a value of type *real*, displays a dialog box which requests the user to enter a *real* value. The parameters RequestStr and Default, of type *string*, allow the programmer to specify the string which requests the information from the user and the default *real* value which appears in the *real* value display box. The *real* dialog box will accept values written in any of distance formats discussed in Unit I. If the user enters an invalid value, the dialog box will produce a beep and re-display the initial default value. Also, if the user selects cancel from the dialog box, the function will return the default value. Statement Example 5-78 demonstrates Function RealDialog in a subprogram. Diagram 5-69 presents the dialog box which is displayed during the execution of Statement Example 5-78.

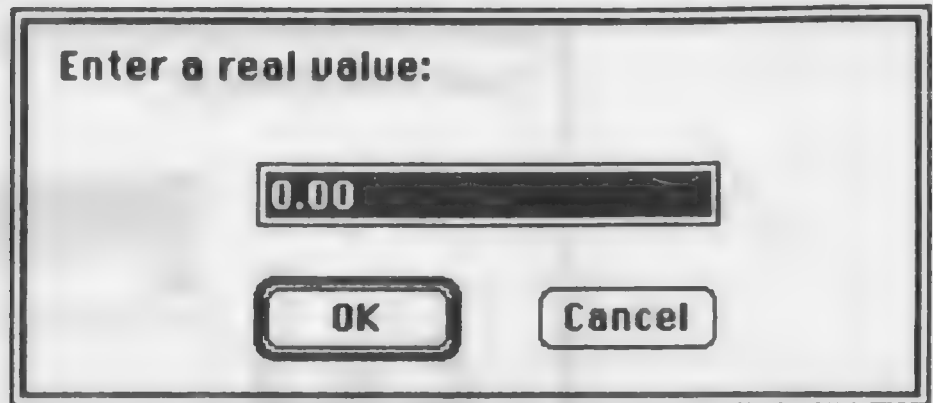
Statement Example 5-78

Function RealDialog Example

```
PROCEDURE RDialog;                {Create subprogram}
VAR
    RealValue : REAL;
BEGIN
    RealValue := REALDIALOG('Enter a real value:', '0.00');
    Writeln('The user entered: ', RealValue:5:2);
END;
RUN(RDialog);
```

Diagram 5-69

Dialog Box from Statement Example 5-78



Function StrDialog(RequestStr,Default : STRING) :
STRING

Function StrDialog, which returns a value of type *string*, displays a dialog box which requests the user to enter a *string* value. The parameters RequestStr and Default, of type *string*, allow the programmer to specify the string which requests the information from the user and the default *string* value which appears in the *string* value display box. If the user enters an invalid *string* value, the dialog box will produce a beep and re-display the initial default value. Also, if the user selects cancel from the dialog box, the function will return the default value. Statement Example 5-79 demonstrates Function StrDialog in a subprogram. Diagram 5-70 presents the dialog box which is displayed during the execution of Statement Example 5-79.

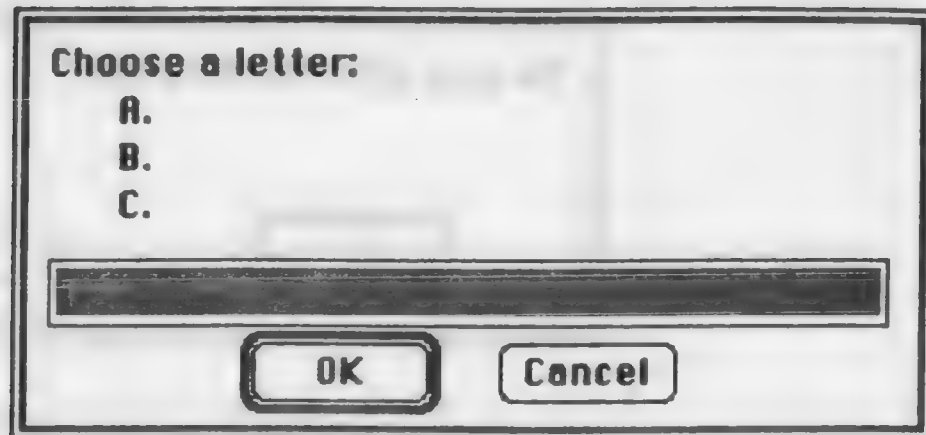
Statement Example 5-79

Function StrDialog Example

```
PROCEDURE StringDialog;           {Create subprogram}
VAR
  StrValue : STRING;              {Declare string variable}
BEGIN
  StrValue := STRDIALOG('Choose a letter:
                        A.
                        B.
                        C.', ' '); {Get string value}
  WRITELN('The user entered: ',StrValue); {Display string value}
END;
RUN(StringDialog);
```

Diagram 5-70

Dialog Box from Statement Example 5-79



Function YNDialog(RequestStr : STRING) : BOOLEAN

Function YNDialog, which returns a value of type *boolean*, displays a dialog box which requests the user to select yes or no. The parameter RequestStr, of type *string*, allows the programmer to specify the string which requests the information from the user. If the user chooses the YES button in the dialog box, the value of YNDialog is true. Conversely, if the user chooses the NO button, the function is false. Statement Example 5-80 demonstrates Function YNDialog in a subprogram. Diagram 5-71 presents the dialog box which is displayed during the execution of Statement Example 5-80.

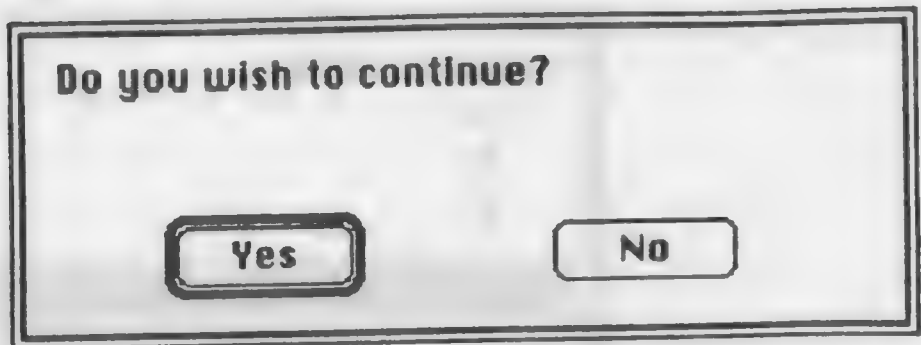
Statement Example 5-80

Function YNDialog Example

```
PROCEDURE YESNO;  
VAR  
    Answer : BOOLEAN;  
BEGIN  
    Answer := YNDIALOG("Do you wish to continue?");  
  
    IF Answer  
    THEN  
        WriteLn('The user wishes to continue');  
    ELSE  
        WriteLn('The user does not wish to continue');  
    END;  
RUN(YESNO);
```


Diagram 5-71

Dialog Box from Statement Example 5-80



Procedure `AlrtDialog(message : STRING);`

Procedure `AlrtDialog` creates a dialog displaying the *string* parameter message and remains there until the user hits the OK button. Subprogram Example 5-5 and Diagram 5-72 demonstrate this routine.

Subprogram Example 5-5

```
PROCEDURE DoAlrt;
```

```
BEGIN
```

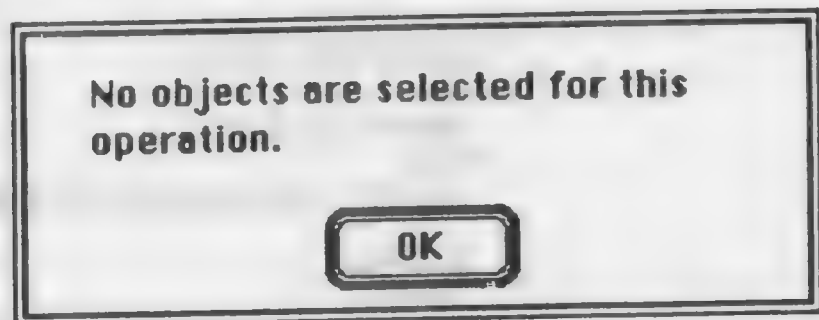
```
    AlrtDialog('No objects are selected for this operation.');
```

```
END;
```

```
RUN(DoAlrt);
```

Diagram 5-72

AlrtDialog Example



Procedure `AngDialog3D(displayStr, xStr, yStr, zStr : STRING; VAR xAngle, yAngle, zAngle : REAL);`

Function `AngDialog3D` displays a dialog box which requests the user to enter three angle values. The parameters `displayStr`, `xStr`, `yStr` and `zStr`, of type *string*, allow the programmer to specify the string which requests the

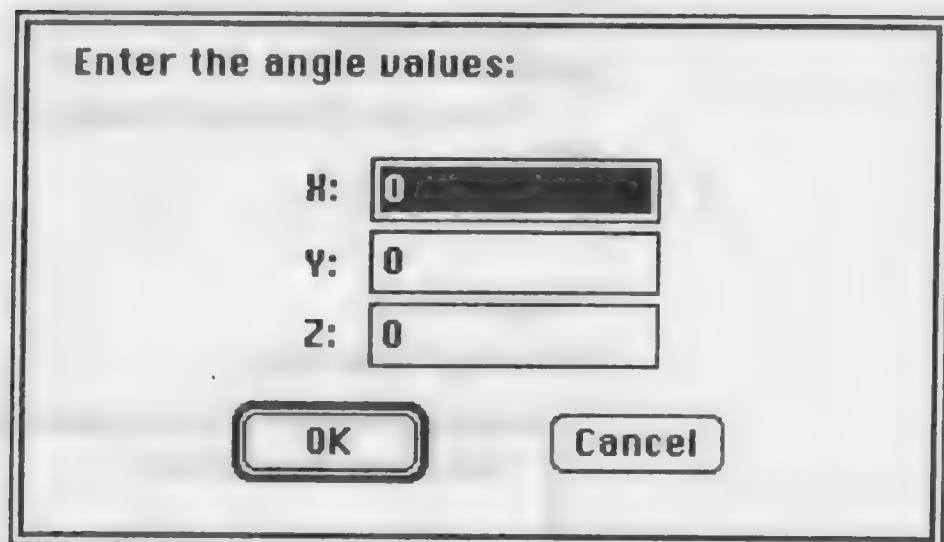
Information from the user and the default angle values which appear in the angle value display boxes. The angle dialog box will accept angle values in any of the angle formats discussed in Unit I. If the user enters an invalid angle value, the dialog box will produce a beep and re-display the initial default value. Also, if the user selects cancel from the dialog box, the function will return the default value. Subprogram Example 5-6 demonstrates Procedure AngDialog3D in a subprogram. Diagram 5-73 presents the dialog box which is displayed during the execution of the example.

Subprogram Example 5-6

```
PROCEDURE Get3DAngles;  
VAR  
    x, y, z : REAL;  
BEGIN  
    AngDialog3D('Enter the angle values:', '0', '0', '0', x, y, z);  
END  
RUN(Get3DAngles);
```

Diagram 5-73

AngDialog3D Example



Enter the angle values:

H: 0

Y: 0

Z: 0

OK Cancel

Procedure PtDialog3D(displayStr, xStr, yStr, zStr : STRING; VAR xPt, yPt, zPt : REAL);

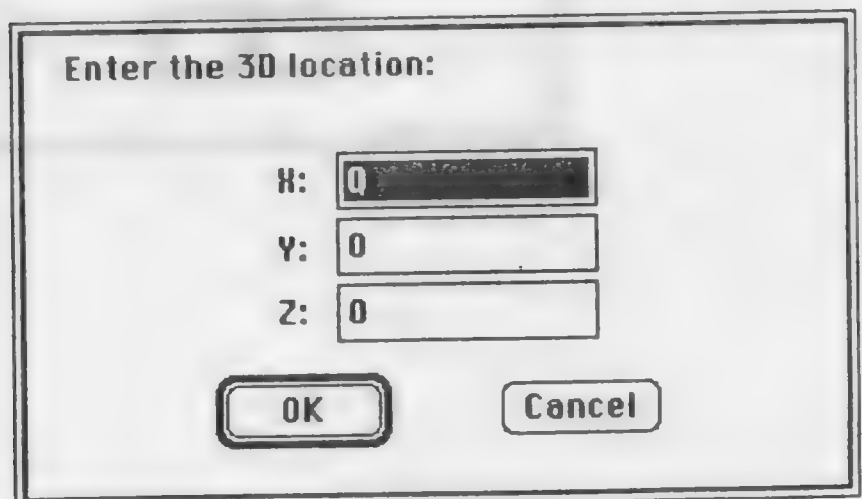
Procedure PtDialog3D displays a dialog box which requests the user to enter a 3D coordinate (point) value. The parameters displayStr, xStr, yStr and zStr of type *string*, allow the programmer to specify the string which requests the information from the user and the default x, y and z values which will appear in the display boxes. The parameters xPt, yPt and zPt, of type *real*, return the numeric coordinate values entered by the user. The 3D point dialog box will accept coordinate values written in any of the coordinate formats discussed in Unit I. If the user enters an invalid coordinate value, the dialog box will produce a beep and re-display the initial default value. Also, if the user selects cancel from the dialog box the function will return the default coordinate values. Subprogram Example 5-7 demonstrates Procedure PtDialog3D in a subprogram. Diagram 5-74 presents the dialog box which is displayed during the execution of the example.

Subprogram Example 5-7

```
PROCEDURE Get3DPts;  
VAR  
    x, y, z : REAL;  
BEGIN  
    PtDialog3D('Enter the 3D location:', '0', '0', '0', x, y, z);  
END  
RUN(Get3DPts);
```

Diagram 5-74

PtDialog3D Example



Enter the 3D location:

X:

Y:

Z:

Function DidCancel : BOOLEAN;

Function DidCancel informs the programmer whether the user clicked the cancel button in the most recent dialog box displayed. Subprogram Example 5-8 demonstrates this routine:

Subprogram Example 5-8

```
PROCEDURE Example;  
  LABEL 1;  
  VAR  
    i : INTEGER;  
  BEGIN  
    i := IntDialog('Enter an integer:', '0');  
    IF DidCancel THEN GOTO 1;  
  1: END;  
  RUN(Example);
```

Section 5-8 Input and Output Routines

Input and output routines allow the programmer to import or export information. These routines include: Procedure Open, Procedure GetFile, Procedure Read, Procedure Readln, Function EOF, Function EOLN, Procedure Rewrite, Procedure Append, Procedure PutFile, Procedure Close, Procedure Write, Procedure Writeln, Procedure Tab and Procedure Space.

MiniPascal contains routines which allow the user to read data from a text file. Prior to being able to read information from a text file, the file must be accessed (opened) with one of the following routines:

PROCEDURE Open(filename : Text);

Procedure Open opens a text file for reading capability. The parameter filename, of type *text*, specifies the name of the text file to be opened. The text file whose name is specified must be located on the same volume as the MiniCad+ application. (Note: The Procedure Close must be called to close the opened text file after it is used.)

Subprogram Example 5-9

```
PROCEDURE ATest;
VAR
    a,b,c : INTEGER;
BEGIN
    Open('MyData');
    Read(a,b,c);
    Close('MyData');
END;
RUN( ATest );
```

PROCEDURE GetFile(VAR filename : Text);

Procedure GetFile displays the standard Macintosh file dialog which requests the user to select a text file. The name of the text file selected is returned in the variable parameter filename. This file routine allows the user to select text files on any available volume. (Note: The programmer should call the function DidCancel after GetFile to insure that the user did not cancel the file selection process.)

Subprogram Example 5-10

```
PROCEDURE ATest;
VAR
    a,b,c : INTEGER;
    fileName : TEXT;
BEGIN
    GetFile(fileName);
    IF NOT DidCancel THEN BEGIN
        Read(a,b,c);
        Close(fileName);
    END;
END;
RUN( ATest );
```

The following routines allow the programmer to read information from a text file.

```
PROCEDURE Read(VAR v1,v2,...vn : INTEGER or
                LONGINT or REAL or CHAR or STRING);
```

Procedure Read reads data from a currently opened text file. The programmer specifies the type of information to be read by the type of the variable parameters provided. The programmer can specify as many variable parameters as desired and of any type listed. The procedure Read will continue to read data from a file in order to fill the value of each variable parameter specified unless an End-Of-File is reached, thereby producing a MiniPascal error. The data in the file must be separated by a carriage-return, tab or space. (Note: if data is encountered which does not correspond to the type of the variable parameter provided, a MiniPascal error is generated.)

For example, suppose the file MyData contains the following text information:

45 8342.34 Hello d

The following subprogram would retrieve the information:

Subprogram Example 5-11

```
PROCEDURE ATest;
VAR
  a : INTEGER;
  b : REAL;
  c : STRING;
  d : CHAR;
BEGIN
  Open('MyData');
  Read(a,b,c,d);
  Close('MyData');
  Writeln('a = ',a);
  Writeln('b = ',b);
  Writeln('c = ',c);
  Writeln('d = ',d);
END;
RUN( ATest );
```

**PROCEDURE ReadLn(VAR v1,v2,...vn : INTEGER or
LONGINT or REAL or CHAR or STRING);**

Procedure ReadLn reads the number of data values specified in the variable parameter list and then automatically sets the reading pointer to the beginning of the next line of data.

For example, suppose the file MyData contains the following text information:

```
45 8342.34 Hello d
37 2541.24 Goodbye f
24 2455.24 SoLong j
```

The following subprogram would retrieve the Integer and Real data:

Subprogram Example 5-12

```
PROCEDURE ATest;
VAR
  a1,a2,a3 : INTEGER;
  b1,b2,b3 : REAL;
BEGIN
  Open('MyData');
  ReadLn(a1,b1);
  ReadLn(a2,b2);
  ReadLn(a3,b3);
  Close('MyData');
END;
RUN( ATest );
```

FUNCTION EOF(filename : Text) : BOOLEAN;

Function EOF returns TRUE if the reading pointer of an open file has reached the end of the file.

Subprogram Example 5-13

```
PROCEDURE ATest;
VAR
  a : INTEGER;
  b : REAL;
  c : STRING;
  d : CHAR;
BEGIN
  Open('MyData');
  WHILE NOT EOF('MyData') DO
    Read(a,b,c,d);
  Close('MyData');
END;
RUN( ATest );
```

FUNCTION EOLN(filename : Text) : BOOLEAN;

Function EOLN returns TRUE if the reading pointer of an open file has reached a carriage return within the file.

Subprogram Example 5-14

```
PROCEDURE ATest;
VAR
  a : INTEGER;
  b : REAL;
  c : STRING;
  d : CHAR;
BEGIN
  Open('MyData');
  WHILE NOT EOLN('MyData') DO
    Read(a,b,c,d);
  Close('MyData');
END;
RUN( ATest );
```

The following routines allow the programmer to write information to a text file:

Procedure Rewrite(FileName : Text);

Procedure Rewrite creates a new file or opens an existing one. The parameter FileName, of type *text*, represents the name of the new file to be created or the name of the existing file to be opened. When procedure Rewrite is called, all future output information is sent to its specified file name. If FileName is the name of an existing file, any information previously in that file is erased prior to any new information being added to it. The default file if no Rewrite statement is encountered is the file named "Output File". Rewrite may not be called while another file is currently open. If this is the case, the currently open file will be closed prior to the new or existing file being opened.

```
PROCEDURE PutFile(commentStr, fileDefault : STRING;  
VAR filename : Text);
```

Procedure PutFile displays the standard Macintosh file dialog which requests the user to select a text file for output. The parameter commentStr displays the provided string in the dialog box, the parameter fileDefault displays the provided string as the default output file name. The name of the text file selected is returned in the variable parameter fileName. This file routine allows the user to select or create output text files on any available volume. (Note: The programmer should call the function DidCancel after GetFile to insure that the user did not cancel the file selection process.)

Subprogram Example 5-15

```
PROCEDURE ATest;  
VAR  
    a,b,c : INTEGER;  
    fileName : TEXT;  
BEGIN  
    PutFile('Output data to the file?', 'The output', fileName);  
    IF NOT DidCancel THEN BEGIN  
        Write('A test.');  
        Close(fileName);  
        END;  
END;  
RUN( ATest );
```


PROCEDURE Append(filename : Text);

Procedure Append allows the user to add text information to the end of a text file. This routine differs from procedure Rewrite in that it does not erase previous text information. The parameter filename, which is of type *text*, is the name of a currently existing text file.

Subprogram Example 5-16

```
PROCEDURE ATest;  
BEGIN  
    Rewrite('MyFile');  
    WriteLn('Write something to MyFile.');
```

Close('MyFile');

```
    Append('MyFile');  
    WriteLn('Add something to MyFile.');
```

Close('MyFile');

```
END;  
RUN( ATest );
```

Procedure Close(FileName : Text);

Procedure Close closes a currently open file. The parameter, FileName, of type *text*, represents the name of a currently opened file. When a read or write file routine is called to open a text file, a corresponding Close statement should be made. If Procedure Close is not called, a MiniPascal error occurs.

Procedure Write(parameter₁, parameter₂, ..., parameter_n);

Procedure Write is one of the fundamental routine calls which specifically writes one or more values to a text file. Each parameter may be a constant, a variable of type *char*, *string*, *real*, *integer*, *longint* or *boolean* or an expression.

Each Write parameter has the following form:

Parameter :< MinWidth> :< DecPlaces>

where the fields MinWidth and DecPlaces are optional. MinWidth specifies the minimum field width. Its value must be greater than or equal to zero. When the MinWidth field is used when writing to a text file, exactly MinWidth characters are written (using leading spaces if necessary), unless the parameter has a value that must be represented in more than MinWidth characters. In such a case, the exact number of characters needed are written. MinWidth can be used with a parameter of any type.

DecPlaces specifies the number of decimal places to be used in the fixed-point representation of a *real* value. It can only be specified if the parameter is of type *real* and if MinWidth is also specified. DecPlaces must be greater than zero. If DecPlaces is not specified and the parameter is of type *real*, a floating-point representation (scientific notation) is written.

Write With a Char Parameter

If Write is given a variable of type *char* and MinWidth is not specified, the character value of the parameter is written to the specified file. If MinWidth is included, exactly MinWidth-1 spaces are written, followed by the character value of the parameter. For example,

```
PROCEDURE WriteExample;  
VAR  
    Ch : CHAR;  
BEGIN  
    Ch := 'C';  
    WRITE('Ch = ',Ch:4);  
END;  
RUN(WriteExample);
```

the preceding subprogram writes out the character 'C' with three spaces before it.

Write With an Integer Parameter

If Write is given a variable of type *integer* and MinWidth is not specified, the *integer* value is written to the specified file. If MinWidth is included and the parameter's length is less than its value, leading spaces will be written to the left of the number. If MinWidth is included and the parameter's length is greater than its value, the entire number is written out. For example,

```
PROCEDURE WriteExample;  
VAR  
    Int : INTEGER;  
BEGIN  
    Int := 363;  
    WRITE('Int = ',Int:2);  
END;  
RUN(WriteExample);
```

the preceding subprogram writes out the numeric value 'Int = 363'.

Write With a Value of Type Real

If the parameter is of type *real*, its decimal representation is written to a text file. This representation depends upon the value of the DecPlaces field (if it is present).

If DecPlaces is present, a fixed-point representation is written. If DecPlaces is absent, a floating-point representation is written. These two cases are discussed below.

Fixed-point representation

If DecPlaces is present when writing out a value of type *real*, DecPlaces' value determines the number of decimal places which will be written out to a text file. If the parameter's decimal portion is greater than DecPlaces, the remaining portion is rounded to the nearest integer (5 being rounded up). If the parameter's decimal portion is less than DecPlaces, zeros are placed in the remaining positions.

```
PROCEDURE WriteExample;  
VAR  
    RealValue : REAL;  
BEGIN  
    RealValue := 36.45;  
    WRITE('RealValue = ',RealValue:5:1);  
END;  
RUN(WriteExample);
```

The result written to a text file from the above subprogram would be 'RealValue = 36.5'.

Floating-point representation

Floating-point representation occurs when DecPlaces is not present. If MinWidth is specified, MinWidth's value represents the number of floating-point digits displayed. The default representation is nine digits.

```
PROCEDURE WriteExample;  
VAR  
    RealValue : REAL;  
BEGIN  
    RealValue := 36.45;  
    WRITE('RealValue = ',RealValue:5);  
END;  
RUN(WriteExample);
```

The result written to a text file from the above subpro-

Write With a String Parameter

The results of using Write with a string variable depend upon the length of the string parameter and whether or not MinWidth is specified. If MinWidth is specified and the length of the parameter is less than MinWidth, then the string is written preceded by a number of spaces equal to MinWidth minus the length of the string. If MinWidth is specified and the length of the parameter is greater than MinWidth, then the first MinWidth number of characters is written. Finally, if MinWidth is specified and the length of the parameter equals MinWidth, or if MinWidth is not specified, the entire string value is written out to the text file.

```
PROCEDURE WriteExample;  
VAR  
    StrValue : STRING;  
BEGIN  
    StrValue := 'String Output';  
    WRITE('StrValue = ', StrValue:5);  
END;  
RUN(WriteExample);
```

The result written to a text file from the above subprogram would be 'StrValue = Strin'.

Write With a Boolean Parameter

If the value of the parameter is type *boolean*, the string 'TRUE' (with a leading space) or the string 'FALSE' is written to the text file. The default value of MinWidth is five. If MinWidth is greater than five, leading spaces are added; if MinWidth is less than five, the character 'T' or 'F' is written, padded with spaces.

```
PROCEDURE WriteExample;  
VAR  
    BoolValue : BOOLEAN;  
BEGIN  
    BoolValue := TRUE;  
    WRITE('BoolValue = ', BoolValue:4);  
END;  
RUN(WriteExample);
```

The result written to a text file from the above subprogram would be ' T'.

Procedure Writeln(parameter₁, parameter₂, ...parameter_n);

Procedure Writeln is an extension of Write. It performs the same actions and then writes a carriage return to the text file. The parameters are the same as those used with Write, except that the Writeln parameters can be omitted. If procedure Writeln is written without any parameters, only a carriage return is written to the text file.

The following routines generate tab and space characters:

PROCEDURE Tab(numOfTabs : INTEGER);

Procedure Tab writes a tab character to the current write file. The parameter numOfTabs specifies the number of tab characters to be written to the file.

PROCEDURE Space(numOfSpaces : INTEGER);

Procedure Space writes a space character to the current write file. The parameter numOfSpaces specifies the number of space characters to be written to the file.

Section 5-9 Menu Routines

There are several MiniPascal routines which allow the programmer to change or execute menus within the MiniCad+ application. These include: Procedure ForEachObject, Procedure Run and Procedure DoMenu.

Procedure ForEachObject (ProcedureHandle: HANDLE ; Search Criteria: : STRING)

Procedure ForEachObject will repeatedly call a procedure passed to it for each object which fits the search criteria

Subprogram Example 5-17

```
PROCEDURE PickRect;  
PROCEDURE Pick (H : HANDLE);  
BEGIN  
    SetSelect(h);  
END;  
BEGIN  
    ForEachObject (Pick, T=RECT);  
END;
```

Procedure Run(subprogramName)

Procedure Run receives a name to an existing subprogram and executes it. Note that Procedure Run cannot be called from within another subprogram. Subprogram Example 5-19 demonstrates this routines.

Subprogram Example 5-19

```
PROCEDURE RunTest;  
BEGIN  
    Sysbeep;  
    Sysbeep;  
    Sysbeep;  
END;  
Run(RunTest);
```

Procedure DoMenu(menuItem,modifierKey);

Procedure DoMenu allows the programmer to execute menu items available in the MiniCad+ application. For example, if the programmer wanted to join two line segments together, he could use the following statement line:

DoMenu(MJoin,NoKey);

The first parameter is a pre-defined constant which refers to a specific menu item (these constants will be discussed shortly). The second parameter specifies whether the programmer wishes to simulate a modifier key (Shift, Command, or Option) being depressed. For example, in normal operation, the user must depress the option key and then select the duplicate menu item in order to view the duplicate array dialog box. The programmer can bring up the same dialog box by entering the following statement in his code:

DoMenu(MDuplicate,OptionKey);

The modifier key constants are as follows:

- ShiftKey:** simulates the shift key being depressed.
- CommandKey:** simulates the command key being depressed.
- OptionKey:** simulates the option key being depressed.
- NoKey:** simulates no key being depressed.

The menu item constants are as follows:

File Menu

MNew
MOpen
MClose
MSave
MSaveAs
MRevert
MImprtPICT
MImprtPICP
MImprt3D
MImprtDXF
MExprtPICT
MExprtTEXT
MExprtDXF
MExprtPS
MPageSetup
MPrint
MQuit

Edit Menu

MUndo
MCut
MCopy
MPaste
MClear
MDuplicate
MSelectAll
MReshape
MNone
MBezier
MCubic
MMirror
MScaleObj

Tool Menu

MDimH
MDimV
MDiagonal
MAngular
MWitLines
MAddSurf
MClipSurf
MIntSurf
MCnvertLine
MCnvertPoly
MCombineSf
MJoin
MTrim
MFillet
MIntersect
MHatch

== Menu

MLayers
MGroup
MUngroup
MEntGroup
MExitGroup
MTopLevel
MAssignCl
MNameObj
MClasses
MCreateSym
MSelectSym
MSym2Group
MEditFlds

Text Menu

MLeftJus
MCentJus
MRightJus
MSingleSp
M1nHalfSp
MDoubleSp

MLowerCase
MUpperCase
MTitleCaps

Page Menu

MNormal
MFit
MSaveView
MSnap2Loc
MSetGrid
MSetOrigin
MScale
MUnits
MDrwScale
MThinLines
MTool
MConstrain

Δ Menu

MAlignGrid
MAlignObjs
MRotateV
MOnScreenV
MFrntView
MBackView
MLeftView
MRightView
MTopView
MBotView
MRotateO
MOnScreenO
MToppleFwd
MToppleBwd
MSwingR
MSwingL
MExtrude
MSweep
MMExtrude
MCon2Mesh
MCon23D
MRender

MSetPersp
MPref
MLock
MUnlock

Color Menu

MPutDown
MPickUp
MUseLayCol
MUseBW

// Menu

MSetThick

\$ Menu

MOpenSprd
MCloseSprd
MPasteAtt
MPasteFunc
MNumber
MAlign
MBorder
MColWidth
MRecalc

Section 5-10 Inquiry Routines

MiniPascal provides several routines which allow the user to obtain specific information from graphic objects within a MiniCad+ file. These routines include: Function CellString, Function CellValue, Function Angle, Function Area, Function Count, Function Height, Function Length, Function ObjectType, Function Perim, Procedure SelectObj, Procedure DSelectObj, Function Width, Function XCenter, Function YCenter, Function LeftBound, Function TopBound, Function RightBound and Function BotBound.

Using Inquiry Routines

Every graphic object within a MiniCad+ file contains specific attributes (characteristics) which identify it from, or with, other graphic objects. These characteristics include:

1. *Object Name:*

An object name is a string of characters which is associated with one particular graphic object. It is assigned to an object by the procedure call NameObject or the Data Palette. Object name strings cannot be greater than twenty characters.

2. *Class Name:*

A class name is a string of characters which is associated with *one or more* graphic objects. It is assigned to an object by the procedure call NameClass or the Data Palette. Class name strings cannot be greater than twenty characters.

3. *Layer Location:*

A graphic object's layer location can distinguish it from other graphic objects. Layer name and locations are designated through the use of the procedure call Layer or by the menu item entitled, "Layer." The name of the layer cannot be greater than twenty characters.

4. *Fill Pattern:*

Each graphic object may have a fill pattern (excluding line segments). Fill patterns are assigned to objects by the "Fill" menu item.

5. *Line Weight:*

Each graphic object may have a particular line weight (pen width). Line weight is determined through the "Lines" menu item.

6. *Line Style:*

Each graphic object may have a particular line style (pen pattern). Pen patterns are determined through the use of the option key and the "Fill" menu.

7. *Object Type:*

Each graphic object is of a particular type (e.g. RECT, OVAL, RRECT, ARC, etc.). These types are determined when a graphic object is created.

8. *Symbol Name:*

Each created symbol has a symbol name. This string, which is twenty characters or less, is designated when a symbol is created.

Through the use of attributes, a graphic object or a set of graphic objects can be identified from others in the same MiniCad+ file. For example, type in Statement Example 5-81 and import the text file into MiniCad+. The graphic display is shown in Diagram 5-75.

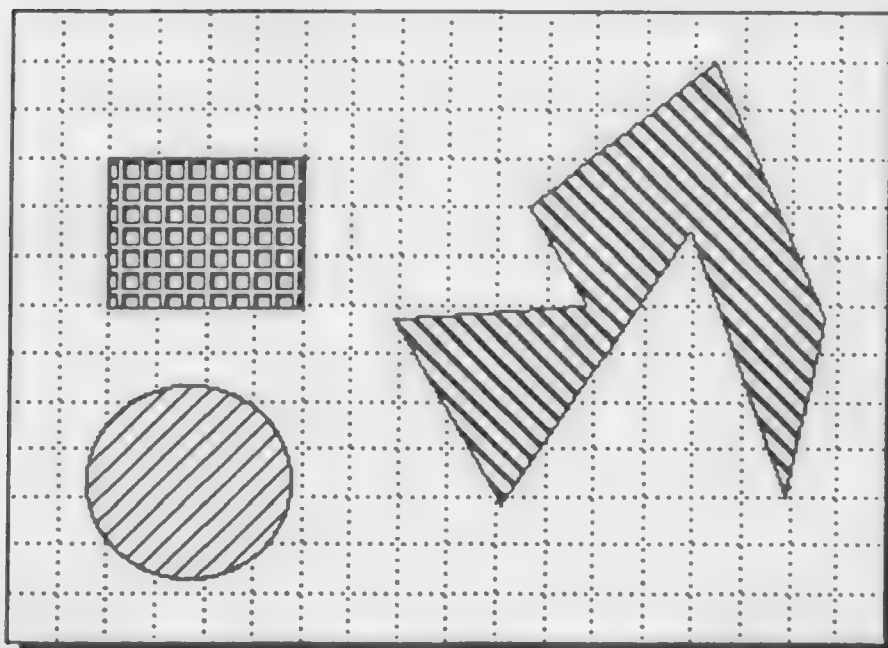
Statement Example 5-81

Inquiry Routine Example

```
NAMECLASS('Edged');  
NAMEOBJECT('Rectangle Object');  
FILLPAT(45);  
RECT(2 119/128",-4 77/512",3 29/32",-4 929/1024");  
  
NAMEOBJECT('Polygon Object');  
FILLPAT(13);  
POLY(  
  5 15/512",-4 101/256",  
  5 245/256",-3 339/512",  
  6 531/1024",-4 979/1024",  
  6 331/1024",-5 55/64",  
  5 855/1024",-4 529/1024",  
  4 929/1024",-5 465/512",  
  4 379/1024",-4 979/1024",  
  5 165/512",-4 113/128",  
  5 15/512",-4 101/256");  
  
NAMECLASS('Rounded');  
NAMEOBJECT('Oval Object');  
FILLPAT(24);  
OVAL(2 213/256",-5 305/1024",3 439/512",-6 281/1024");
```

Diagram 5-75

Inquiry Example Output



Each object in the graphic display has its own set of attributes. Some of these attributes differ from the other objects on the screen and some are similar. Table 5-12 presents some of these attributes.

Table 5-12
Object Attributes

<i>Object Name</i>	<i>Class Name</i>	<i>Fill Pattern</i>	<i>Object Type</i>	<i>Layer Location</i>	<i>Line Weight</i>	<i>Line Style</i>
Rectangle Object	Edged	#45	RECT	Untitled	14	2
Polygon Object	Edged	#13	POLY	Untitled	14	2
Oval Object	Rounded	#24	OVAL	Untitled	14	2

As you can see from Table 5-12, some attributes are the same (Layer Location, Line Weight and Line Style) while others differ (Object Name, Class Name, Fill Pattern and Object Type). Inquiry routines are used in order to perform specific operations on specific graphic objects. How are the specific graphic objects identified? As you probably guessed, through attributes.

Let's perform a simple operation on the graphic objects created in Statement Example 5-81. Let's find out the number of objects which are in the class "Edged." Subprogram Example 5-20 presents a routine which will accomplish this. With the graphic example shown in Diagram 5-75, the answer to this question is rather obvious and it may appear that such a routine is not needed. However, the answer would not be as obvious for a MiniCad+ file which reflected an entire house and contained thousands of graphic objects.

Subprogram Example 5-20

Inquiry Routine Example

```
PROCEDURE GetNum;  
VAR  
    NumOfEdged : INTEGER;  
BEGIN  
    NumOfEdged := COUNT(C='Edged');  
    WRITELN('The number is : ',NumOfEdged);  
END;  
RUN(GetNum);
```

Specifically, the number of objects which are in the class 'Edged' is returned by the function Count. Let's examine inquiry routines in general by examining the components of this routine.

Inquiry routines are composed of two parts: a routine call and search criteria.

Routine Call Search Criteria
 ↓ ↓
 └──────────┘
 COUNT(C='Edged');

The routine call, such as COUNT, is merely the name of the procedure or function that we plan to use. This name specifies precisely what type of action we wish to perform. In Subprogram Example 5-20, we wanted the computer to count particular objects in a file, thus we specified the routine COUNT (other routine calls will be discussed shortly). The second part of an inquiry routine, the search criteria, determines what specific objects should be included in the specified action. In the above example, the computer was told to include all objects which were in the class 'Edged.'

The search criteria in Subprogram Example 5-20 contained only one attribute, class. However, the search criteria of an inquiry routine can be as complex as desired in order to perform a particular operation on specific objects. Thus, in the above example, we could have had the computer count all objects which are in class 'Edged,' which have the object name 'Polygon Object' and who have fill pattern #13. This search criteria would be represented as follows:

```
COUNT((C='Edged')and( N='Polygon Object')and( FP =13));
```


In the previous example there are three attributes listed in the search criteria. Each attribute may be broken down into two parts which are separated by a colon: the attribute type and the attribute field.

Attribute Type Attribute Field
 ↓ ↓
 C = 'Edged'

The attribute type specifies what characteristics of an object the computer will be checking. Therefore, in the above example, the first attribute type, class, tells the computer to look at each object's class characteristic. The attribute field provides the information that an object must have in order for the inquiry routine action to be performed on it (in the example, 'Edged').

When calculating inquiry routines, the computer checks each graphic object in a MiniCad+ file to see if it has all of the attributes listed in the search criteria. If an object does not meet all of the search criteria attributes, it will not be included in the inquiry operation. Therefore, if we replaced the complex example shown above into Subprogram Example 5-20 the answer would be one.

There may be times when the programmer may wish to simultaneously identify objects which have different attribute fields. For example, suppose the user wanted to count all objects in Diagram 5-75 which contained either fill pattern #13 or fill pattern #24. The search criteria information would be written as follows

```
COUNT(FP =(13,24));
```

The vertical bar symbol tells the computer that the attribute type of an object can contain more than just one value in order to pass the criteria test. If this inquiry routine replaced the one in Subprogram Example 5-20 the result would be two.

Search criteria in an inquiry routine performs similar to boolean expressions (which were discussed in Unit III). When the programmer specifies more than one attribute field in the search criteria of an inquiry routine, it is as if the user was placing an 'OR' operator between the attribute fields. Therefore, the above example may be viewed as: Count all objects which have either fill pattern #13 or fill pattern #24. The result of the test is true if the object contains either one of these fill patterns.

Also, when using more than one attribute type in the search criteria of an inquiry routine, it is as if the user was placing an 'AND' operator between each attribute type. Let's look at one of the previous examples again.

```
COUNT((C='Edged')and( N='Polygon Object')and( FP=13));
```

The above example is telling the computer that in order for an object to be included in the count operation, it must be in the class 'Edged' and it must have the object name 'Polygon Object' and it must contain fill pattern #13. The result of the test is true only if the currently tested object meets all of the search criteria.

Attribute types and fields have specific identifiers when used in the search criteria of an inquiry routine. The identifiers are shown in Table 5-13 and Table 5-14.

Table 5-13

Attribute Type Identifiers

<i>Attribute Type</i>	<i>Attribute Type Identifier</i>	<i>Example</i>
Arrowhead	AR	(AR=1)
Class Name	C	(C='Tile')
Every Object	All	All
Fill Background	FB	(FB=White)
Fill Foreground	FF	(FF=Black)
Fill Pattern	FP	(FP=3)
Layer Location	L	(L='Basement')
LineWeight	LW	(LW=2)
LineStyle	LST	(LST=2)
Object Name	N	(N='Brick')
Object Record	R	(R ='Doors')
Object Type	T	(T='Rect')
Pen Background	PB	(PB=Black)
Pen Foreground	PF	(PB=Black)
Pen Pattern	PP	(PP=1)
Selected status	Sel	(Sel='Select')
Symbol Name	S	(S='Window')
Visibility	V	(V=True)
In Symbol	In Symbol	In Symbol

Table 5-14

Attribute Field Identifiers

<i>Attribute Field</i>	<i>Attribute Field Identifier</i>	<i>Example</i>
Object Name	String of 20 or less characters	(N= Brick)
Class Name	String of 20 or less characters	(C= Tile)
Layer Location	String of 20 or less characters	(L= Basement)
Fill Pattern	FP followed by fill pattern number	(FP=3)
LineWeight	LW followed by line weight number	(LW=2)
LineStyle	LS followed by pen pattern number	(LST=2)
Object Type:		
Rectangle	'Rect'	(T='Rect')
Oval	'Oval'	(T='Oval')
Polygon	'Poly'	(T='Poly')
Arc	'Arc'	(T='Arc')
Quarter Arc	'QArc'	(T='QArc')
Line	'Line'	(T='Line')
Text	'Text'	(T='Text')
Rounded Rectangle	'RRect'	(T='RRect')
Locus	'Locus'	(T='Locus')
Free Hand Line	'FHand'	(T='FHand')
Symbol	'Symbol'	(T='S')
Sweep Object	'Sweep'	(T='Sweep')
Multiple Extrude Object	'MXtrd'	(T='MXtrd')
SpreadSheet	'SprdSheet'	(T='SprdSheet')
Group	'Group'	(T='Group')
Mesh Object	'Mesh'	(T='Mesh')
Extruded Object	'Xtrd'	(T='Xtrd')
Poly3D Object	'Poly3D'	(T='Poly3D')
Symbol Name	String of 20 or less characters	(S= Window')
Selected Status		(Sel = True)
Selected Status		(Sel = False)

Inquiry Routines

All of the inquiry routines, excluding Function Eval and Function EvalString, must receive search criteria information in order to obtain information for the programmer. Search criteria information may include any of the attribute types or attribute fields which were described previously.

Function CellString(Row, Column : INTEGER) : STRING;

Function CellString returns the string value contained within the active spreadsheet cell designated by the parameters Row and Column. If the specified cell does not contain a string value, an error is generated. Subprogram Example 5-21 demonstrates Function CellString.

Subprogram Example 5-21

Function CellString Example.

```
PROCEDURE CellEx;  
VAR str : STRING;  
BEGIN  
    SPRDSHEET(0,0,3,3);  
    LOADCELL(1,1,'Cell 1,1');  
    str := CELLSTRING(1,1);  
    WRITELN('str = ',str);  
END;  
RUN(CellEx);
```

Function CellValue(Row, Column : INTEGER) : REAL;

Function CellValue returns the numeric value contained within the active spreadsheet cell designated by the parameters Row and Column. If the specified cell does not contain a numeric value, an error is generated. Subprogram Example 5-22 demonstrates Function CellValue.

Subprogram Example 5-22

Function CellValue Example.

```
PROCEDURE CellEx;  
VAR num : REAL;  
BEGIN  
    SPRDSHEET(0,0,3,3);  
    LOADCELL(1,1,'= 4.23 * 7');  
    num := CELLVALUE(1,1);  
    WRITELN('num = ',num:10:2);  
END;  
RUN(CellEx);
```

Function Angle(<Search Criteria>) : REAL

Function Angle, which returns a *real* result, provides the angle value of a line segment or an arc. If an object matches the search criteria but is not a line segment or an arc, the value zero is returned. Also, if more than one line segment or arc matches the search criteria, the function will return the sum of the angle values. Subprogram Example 5-23 demonstrates this routine.

Subprogram Example 5-23

Function Angle Example

```
PROCEDURE AngEx;
VAR
    AngleValue : REAL;
BEGIN
    AngleValue := ANGLE(N='LineSeg');
    Writeln('AngleValue = ',AngleValue);
END;
RUN(AngEx);
```

Function Area(<Search Criteria>) : REAL

Function Area, which returns a *real* result, provides the area of an object. If more than one object matches the search criteria, the function will return the sum of the objects' areas. Subprogram Example 5-24 demonstrates this routine.

Subprogram Example 5-24

Function Area Example

```
PROCEDURE AreaEx;
VAR
    AreaValue : REAL;
BEGIN
    AreaValue := AREA(N='Plywood')and(L='First');
    Writeln('AreaValue = ',AreaValue);
END;
RUN(AreaEx);
```

Function Count(<Search Criteria>) : LONGINT

Function Count, which returns a *longint* value, counts all of the objects which match the search criteria provided. Subprogram Example 5-25 demonstrates this routine.

Subprogram Example 5-25

Function Count Example

```
PROCEDURE CountEx;
VAR
    CountValue : LONGINT;
BEGIN
    CountValue := COUNT(FP=4')and(T='Rect');
    Writeln('CountValue = ',CountValue);
END;
RUN(CountEx);
```

Function Height(<Search Criteria>) : REAL

Function Height, which returns a *real* result, provides the height of an object. If more than one object matches the search criteria, the function will return the sum of the objects' heights. Subprogram Example 5-26 demonstrates this routine.

Subprogram Example 5-26

Function Height Example

```
PROCEDURE HeightEx;
VAR
    HeightValue : REAL;
BEGIN
    HeightValue := HEIGHT(N='Wall');
    Writeln('HeightValue = ',HeightValue);
END;
RUN(HeightEx);
```

Function Length(<Search Criteria>) : REAL

Function Length, which returns a *real* result, provides the length of an object. If more than one object matches the search criteria, the function will return the sum of the objects' lengths. Subprogram Example 5-27 demonstrates this routine.

Subprogram Example 5-27

Function Length Example

```
PROCEDURE LengthEx;  
VAR  
    LengthValue : REAL;  
BEGIN  
    LengthValue := LENGTH(N='Board');  
    Writeln('LengthValue = ',LengthValue);  
END;  
RUN(LengthEx);
```

Function ObjectType(<Search Criteria>) : INTEGER

Function ObjectType, which returns an *integer* result, provides an object's numeric type. These numeric types are presented in Table 5-15. If more than one object matches the search criteria, the numeric type of the last object encountered will be the returned value.

Table 5-15

Object Type Numeric Values

<i>Object Type</i>	<i>Numeric Value</i>
Line	2
Rect	3
Oval	4
Poly	5
Arc	6
FHand	8
Text	10
Group	11
QArc	12
RRect	13
Symbol	15
Locus	17
SprdSheet	18
Xtrd	24
Poly3D	25
Sweep	34
MXtrd	38
Mesh	40

Subprogram Example 5-28 demonstrates Function `ObjectType`.

Subprogram Example 5-28

Function `ObjectType` Example

```
PROCEDURE ObjTypeEx;
VAR
    ObjTypeValue : INTEGER;
BEGIN
    ObjTypeValue := OBJECTTYPE(N='Wall');
    Writeln('ObjTypeValue = ', ObjTypeValue);
END;
RUN(ObjTypeEx);
```

Function `Perim (<Search Criteria>) : REAL`

Function `Perim`, which returns a *real* result, provides the perimeter of an object. If more than one object matches the search criteria, the function will return the sum of the objects' perimeters. Subprogram Example 5-29 demonstrates this routine.

Subprogram Example 5-29

Function `Perim` Example

```
PROCEDURE PerimEx;
VAR
    PerimValue : REAL;
BEGIN
    PerimValue := PERIM(C='Fence');
    Writeln('PerimValue = ', PerimValue);
END;
RUN(PerimEx);
```

Procedure `SelectObj(<Search Criteria>)`

Procedure `SelectObj` selects all objects which match the search criteria. Subprogram Example 5-30 demonstrates this routine.

Subprogram Example 5-30

Procedure SelectObj Example

```
PROCEDURE SelectObjEx;  
BEGIN  
    SELECTOBJ(N='Brick');  
END;  
RUN(SelectObjEx);
```

Procedure DSelectObj(<Search Criteria>)

Procedure DSelectObj deselects all objects which match the search criteria. Subprogram Example 5-31 demonstrates this routine.

Subprogram Example 5-31

Procedure DSelectObj Example

```
PROCEDURE DSelectObjEx;  
BEGIN  
    DSelectObj(S='Window');  
END;  
RUN(DSelectObjEx);
```

Function Width(<Search Criteria>) : REAL

Function Width, which returns a *real* result, provides the width of an object. If more than one object matches the search criteria, the function will return the sum of the objects' widths. Subprogram Example 5-32 demonstrates this routine.

Subprogram Example 5-32

Function Width Example

```
PROCEDURE WidthEx;  
VAR  
    WidthValue : REAL;  
BEGIN  
    WidthValue := WIDTH(C='Box');  
    WriteLn('WidthValue = ', WidthValue);  
END;  
RUN(WidthEx);
```

Function XCenter(<Search Criteria>) : REAL

Function XCenter, which returns a *real* result, provides the x coordinate of the center point of the object. If more than one object matches the search criteria, the function will return the x coordinate of the center point of the last object found. Subprogram Example 5-33 demonstrates this routine.

Subprogram Example 5-33

Function XCenter Example

```
PROCEDURE XCenEx;
VAR
    XCenValue : REAL;
BEGIN
    XCenValue := XCENTER(N='Board');
    Writeln('XCenValue = ',XCenValue);
END;
RUN(XCenEx);
```

Function YCenter(<Search Criteria>) : REAL

Function YCenter, which returns a *real* result, provides the y coordinate of the center point of the object. If more than one object matches the search criteria, the function will return the y coordinate of the center point of the last object found. Subprogram Example 5-34 demonstrates this routine.

Subprogram Example 5-34

Function YCenter Example

```
PROCEDURE YCenEx;
VAR
    YCenValue : REAL;
BEGIN
    YCenValue := YCENTER(N='Board');
    Writeln('YCenValue = ',YCenValue);
END;
RUN(YCenEx);
```

Function LeftBound(<Search Criteria>) : REAL

Function LeftBound, which returns a *real* result, provides the left (x coordinate) value of the object's upper left corner. If more than one object matches the search criteria, the function will return the left value of the last object found. Subprogram Example 5-35 demonstrates this routine.

Subprogram Example 5-35

Function LeftBound Example

```
PROCEDURE LeftBEx;
VAR
    LeftBValue : REAL;
BEGIN
    LeftBValue := LEFTBOUND(N='MyRect');
    Writeln('LeftBValue = ',LeftBValue);
END;
RUN(LeftBEx);
```

Function TopBound(<Search Criteria>) : REAL

Function TopBound, which returns a *real* result, provides the top (y coordinate) value of the object's upper left corner. If more than one object matches the search criteria, the function will return the top value of the last object found. Subprogram Example 5-36 demonstrates this routine.

Subprogram Example 5-36

Function TopBound Example

```
PROCEDURE TopBEx;
VAR
    TopBValue : REAL;
BEGIN
    TopBValue := TOPBOUND(N='MyRect');
    Writeln('TopBValue = ',TopBValue);
END;
RUN(TopBEx);
```

Function RightBound(<Search Criteria>) : REAL

Function RightBound, which returns a *real* result, provides the right (x coordinate) value of the object's lower right corner. If more than one object matches the search criteria, the function will return the right value of the last object found. Subprogram Example 5-37 demonstrates this routine.

Subprogram Example 5-37

Function RightBound Example

```
PROCEDURE RightBEx;
VAR
    RightBValue : REAL;
BEGIN
    RightBValue := RIGHTBOUND(N='MyRect');
    Writeln('RightBValue = ',RightBValue);
END;
MENUITEM(RightBEx);
```

Function BotBound(<Search Criteria>) : REAL

Function BotBound, which returns a *real* result, provides the bottom (y coordinate) value of the object's lower right corner. If more than one object matches the search criteria, the function will return the bottom value of the last object found. Subprogram Example 5-38 demonstrates this routine.

Subprogram Example 5-38

Function BotBound Example

```
PROCEDURE BotBEx;
VAR
    BotBValue : REAL;
BEGIN
    BotBValue := BOTBOUND(N='MyRect');
    Writeln('BotBValue = ',BotBValue);
END;
RUN(BotBEx);
```

Unit VI Standard Pre-defined Routines

Introduction Standard pre-defined routines aid the user in the calculation of numeric values and the manipulation of data. These routines include: Function Trunc, Function Round, Function Abs, Function Sqr, Function Sin, Function Cos, Function Exp, Function Ln, Function Sqrt, Function Arctan, Function ArcSin, Function ArcCos, Function Rad2Deg, Function Deg2Rad, Function Ord, Function Chr, Function Len, Function Pos, Function Concat, Function Copy, Procedure Delete, Procedure Insert, Function Str2Num, Function Num2Str, Function Num2StrF, Function PtInPoly, Function Distance, Function PtInRect, Function UnionRect, Function EqualRect and Function EqualPt.

Section 6-1 Transfer Functions

Transfer functions transfer a value from an expression of one type to an expression of another type.

Function Trunc(*x* : REAL) : LONGINT

The trunc function converts a *real* value to a *longint* value. Its parameter *x* is an expression with a value of type *real*. Trunc(*x*) returns a *longint* result that is the value of *x* rounded to the largest whole number between 0 and *x*.

Function Round(*x* : REAL) : LONGINT

The Round function converts a *real* value to a *longint* value. Its parameter *x* is an expression with a value of type *real*. If *x* is exactly between two whole numbers, the result is the number with the greatest absolute magnitude.

Section 6-2 Arithmetic Functions

Arithmetic functions perform mathematical operations on specified numeric values.

Function Abs(x)

The Abs function returns the absolute value of a numeric value. Its parameter *x* is an expression with a value of type *real*, *integer*, or *longint*. Abs(*x*) returns the absolute value of *x*, with the same type.

Function Sqr(x)

The Sqr function returns the square of a numeric value. Its parameter *x* is an expression with a value of type *real*, *integer*, or *longint*. Sqr(*x*) returns the square of *x*.

If *x* is of type *real*, the result is *real*; if *x* is of type *longint*, the result is *longint*; and if *x* is of type *integer*, the result may be either *integer* or *longint*.

Function Sin(x) : REAL

The Sin function returns a *real* value that is the sine of a numeric value. Its parameter *x* is an expression with a value of type *real*, *integer*, or *longint*. This value is assumed to represent an angle in radians.

Function Cos(x) : REAL

The Cos function returns a *real* value that is the cosine of a numeric value. Its parameter *x* is an expression with a value of type *real*, *integer*, or *longint*. This value is assumed to represent an angle in radians.

Function Exp(x) : REAL

The Exp function returns a *real* value that is the natural exponential of a numeric value. Its parameter *x* is an expression with a value of type *real*, *integer*, or *longint*. All possible values are valid. Exp(*x*) returns the value of e^x , where *e* is the base of the natural logarithms.

Function Ln(x) : REAL

The Ln function returns a *real* value that is the natural logarithm of a numeric value. Its parameter *x* is an expression with a value of type *real*, *integer*, or *longint*. All nonnegative values are valid; negative values are invalid. If *x* is nonnegative, Ln(*x*) returns the natural logarithm (\log_e) of *x*.

Function Sqrt(x) : REAL

The Sqrt function returns a *real* value that is the square root of a numeric value. Its parameter *x* is an expression with a value of type *real*, *integer*, or *longint*. All nonnegative values are valid; negative values are invalid.

Function Arctan(x) : REAL

The Arctan function returns a *real* value that is the principle value, in radians, of the arctangent of a numeric value. Its parameter *x* is an expression with a value of type *real*, *integer*, or *longint*. All numeric values of *x* are valid.

Function ArcSin(x) : REAL

The ArcSin function returns a *real* value that is the principle value, in radians, of the arcsine of a numeric value. Its parameter *x* is an expression with a value of type *real*, *integer*, or *longint*. All numeric values of *x* are valid.

Function ArcCos(x) : REAL

The ArcCos function returns a *real* value that is the principle value, in radians, of the arccosine of a numeric value. Its parameter *x* is an expression with a value of type *real*, *integer*, or *longint*. All numeric values of *x* are valid.

Function Rad2Deg(RadianValue : REAL) : REAL

Function Rad2Deg returns a *real* value that is the conversion of a radian value to decimal degrees. The parameter RadianValue, of type *real*, represents the radian value to be converted.

Function Deg2Rad(DegreeValue : REAL) : REAL

Function Deg2Rad returns a *real* value that is the conversion of a degree value to a radian value. The parameter DegreeValue, of type *real*, represents the decimal degree value to be converted.

Section 6-3 Ordinal Functions

Ordinal functions operate on the ordinal value of scalar types.

Function Ord(x)

The Ord function returns the ordinal number of a scalar type value. Its parameter *x* is an expression with a value of scalar type.

If *x* is of type *integer* or *longint*, the result is the same as *x*.

For a parameter of type *char*, the result is the corresponding ASCII code. For a parameter of type *boolean*, ord(false) returns 0 and ord(true) returns 1.

Function Chr(x) : CHAR

The Chr function returns the char value corresponding to a whole-number value. Its parameter *x* is an expression with an *integer* or *longint* value. Chr(*x*) returns the char value whose ordinal number (that is, its ASCII code) is *x*, if *x* is in the range 0..255. If *x* is not in the range 0..255, the value returned is not within the range of the type *char*, and any attempt to assign it to a variable of type *char* will cause an error.

Section 6-4 String Procedures and functions

String procedures and functions provide information or perform an operation on a specified string.

Function Len(str : STRING) : INTEGER

The Len function returns an *integer* value that is the current length of its parameter str, which must be of type *string*.

Function Pos(substr, str : STRING) : INTEGER

The Pos function searches for substr within str and returns an *integer* value that is the index of the first character of substr within str. Both parameters must be of type *string*.

Function Concat(str₁, str₂, ..., str_n : STRING) : STRING

Concat concatenates all the parameters in the order in which they are written and returns the concatenated string. Each parameter is an expression with a value of type *string*. Any number of parameters may be passed. Note that the number of characters in the result cannot exceed 255.

Function Copy(source : STRING; index, count : INTEGER) : STRING

Copy returns a string containing count characters from source, beginning at source[index]. The parameter source is an expression with a value of type *string*. The parameter index is an expression with an *integer* value in the range 1..255. The parameter count is an expression with an *integer* value in the range 1..255.

If the values of index or count are out of range or if there are not count characters in source starting at source[index], Copy returns the null string.

Procedure Delete(dest : STRING; index, count : INTEGER)

Delete removes count characters from the value of dest, beginning at dest[index]. The parameter dest is a variable parameter of type *string*. The parameter index is an expression with an *integer* value in the range 1..255. The parameter count is an expression with an *integer* value in the range 1..255.

If the values of `index` or `count` are out of range or if `index` is greater than `Len(dest)`, `Delete` is ignored. If the attempted deletion extends beyond the end of `dest`, `dest` becomes truncated at `index-1`.

**Procedure Insert(source : STRING; VAR dest : STRING;
index : INTEGER)**

The procedure `Insert` inserts `source` into `dest`. The first character of `source` becomes `dest[index]`. The parameter `source` is an expression with a value of type *string*. The parameter `dest` is a variable parameter of type *string*. The parameter `index` is an expression with an *integer* value in the range 1..255.

If the value of `index` is out of range, `Insert` is ignored.

Function Str2Num(x : REAL) : REAL

The function `Str2Num` takes a numeric string argument and converts it to type *real*.

**Function Num2Str(DecimalPlaces : INTEGER;
x : REAL) : STRING**

The function `Num2Str` converts a real value to a string. The argument `DecimalPlaces`, of type *integer*, determines the number of decimal places represented in the string. The argument's value may be between -1 and 9. If `DecimalPlaces` value is -1, the string returned will be the numeric value written in scientific notation.

Function Num2StrF(x : REAL) : STRING

The function `Num2StrF` converts a real value into a string. The numeric value will be displayed in the current unit settings of the drawing.

Procedure UpString(VAR s : STRING);

Procedure `UpString` converts all characters in the parameter `s` to uppercase.

Section 6-5 Graphic Calculation Routines

Graphic calculation routines perform basic graphic numeric operations.

Function PtInPoly(x,y:REAL;h:Handle):BOOLEAN

If h is a handle to a polygon, then Function PtInPoly returns TRUE if the coordinate location x,y is located within the polygon, otherwise it returns FALSE.

Function Distance(x1,y1,x2,y2:REAL):REAL

Function Distance returns the distance value between the two coordinate locations x1,y1 and x2,y2.

Function PtInRect(x,y,x1,y1,x2,y2:REAL):BOOLEAN

Function PtInRect returns TRUE if the coordinate location x,y is located within the rectangular boundary specified by x1,y1,x2,y2, otherwise it returns FALSE.

**Procedure UnionRect(x1,y1,x2,y2,x3,y3,x4,y4:REAL;
VAR x5,y5,x6,y6:REAL)**

Procedure UnionRect receives two rectangular boundaries, specified by x1,y1,x2,y2 and x3,y3,x4,y4, and returns a rectangular boundary which encloses both rectangles.

**Function EqualRect(x1,y1,x2,y2,x3,y3,x4,y4:REAL):
BOOLEAN**

Function EqualRect returns TRUE if the two rectangular boundaries, x1,y1,x2,y2 and x3,y3,x4,y4, are equal, otherwise it returns FALSE.

Function EqualPt(x1,y1,x2,y2:REAL):BOOLEAN

Function EqualPt returns TRUE if the two coordinate locations, x1,y1 and x2,y2, are equal, otherwise it returns FALSE.

Unit VII Using Handle Variables.

Introduction This unit discusses the use of Handle variables within the MiniPascal programming language.

Section 7-1 Handle Variables

Handle variables are used as a means of communication between a MiniPascal subprogram and the graphic objects within a MiniCad+ document. They establish a connection which allows the programmer to either receive or change graphic information within a current file. For example, the programmer may wish to find the endpoints of all line segments in a drawing. Using *handle* variables, the programmer can write a subprogram which will "look at" all objects in a MiniCad+ file. The *handle* connection will tell the programmer what type of object he is looking at, and, if it is a line segment, the specific endpoints of the object. *Handle* variables may also be used to obtain a connection to layers within a MiniCad+ document. They can obtain information such as layer names and layer scales. Pre-defined routines exist for establishing *handle* connections to a MiniCad+ drawing. Once a connection is made, other routines allow the programmer to send or receive information.

A *handle* is data type, therefore the programmer stores handles by declaring variables of type *handle*. For example:

Subprogram Example 7-1

```
PROCEDURE HandleEx;  
VAR  
    h : Handle;  
BEGIN  
    h := FObject;  
END;  
MenuItem(HandleEx);
```

The above procedure establishes a connection between the programming language and the first object in the active file (FObject is a pre-defined handle function). The variable *h* stores the connection to the first object. What happens if Subprogram Example 7-1 was executed and there were no objects in the active drawing? This is a

situation which can easily occur. The programmer must perform a check upon the handle variable prior to using it within other routines to ensure that it is connected to a graphic object or layer. If a handle variable is not connected to any object or layer, then it contains the value NIL. Thus, the programmer would perform the following check:

Subprogram Example 7-2

```
PROCEDURE HandleEx;
VAR
  h : Handle;
BEGIN
  h := FObject;
  IF h <> NIL THEN
    BEGIN
      END;
    END;
END;
RUN(HandleEx);
```

Section 7-2 Handle Routines which Obtain a Connection to an Object, Layer or Symbol Library.

Many pre-defined routines exist which obtain connections to graphic objects, layers or symbol libraries within a drawing. They include:

FUNCTION FObject : Handle;

Function FObject returns a handle to the first object in the active document. If the object does not exist, the function returns NIL.

FUNCTION LObject : Handle;

Function LObject returns a handle to the last object in the active document. If the object does not exist, the function returns NIL.

FUNCTION FActLayer : Handle;

Function FActLayer returns a handle to the first object on the active layer. If the object does not exist, the function returns NIL.

FUNCTION FSActLayer : Handle;

Function FSActLayer returns a handle to the first selected object on the active layer. If the object does not exist, the function returns NIL.

FUNCTION LSActLayer : Handle;

Function LSActLayer returns a handle to the last selected object on the active layer. If the object does not exist, the function returns NIL.

FUNCTION ActSSheet : Handle;

Function ActSSheet returns a handle to the currently active spreadsheet. If no spreadsheet is currently active, the function returns NIL.

FUNCTION GetObject(name : STRING) : Handle;

Function GetObject receives an object name and returns a handle to the object which has the name. If the specified object name does not exist, the function returns NIL.

FUNCTION FLayer : Handle;

Function FLayer returns a handle to the first layer in a drawing. If the first layer does not exist, the function returns NIL.

FUNCTION LLayer : Handle;

Function LLayer returns a handle to the last layer in a drawing. If the last layer does not exist, the function returns NIL.

FUNCTION ActLayer : Handle;

Function ActLayer returns a handle to the currently active layer in a drawing. If an active layer does not exist, the function returns NIL.

FUNCTION FSOBJECT(h : Handle) : Handle;

Function FSOBJECT returns a handle to the first selected object on the layer specified by the handle h. If the first selected object does not exist, the function returns NIL.

FUNCTION GetLayer(h : Handle) : Handle;

Function GetLayer receives a handle to a graphic object and returns a handle to its corresponding layer.

FUNCTION FSymDef : Handle;

Function FSymDef returns a handle to the first object in the current document's symbol library. If there are no objects in the symbol library, the function returns NIL.

MiniCad+ allows graphic objects to be nested within each other. For example, a grouped object is composed of several internal graphic objects. The user can access these internal objects by using the Enter Group menu command. Likewise, the programmer can access these internal objects through the use of handle variables.

Every grouped object, symbol definition or three-dimensional object stores its own internal objects in a list stemming from the object. A grouped object has a list which contains the objects in that particular group. A symbol definition has a list which contains the objects in that particular symbol definition. Finally, a three-dimensional object has a list which contains the two- and three-dimensional objects which compose the three-dimensional object. All of these object lists can be accessed through pre-defined routines.

FUNCTION FInGroup(h :Handle) : Handle;

Function FInGroup receives a handle to a grouped object and returns a handle to the first object within the group. If an internal object does not exist, the function returns NIL.

FUNCTION FIn3D(h : Handle) : Handle;

Function FIn3D receives a handle to a three-dimensional object and returns a handle to the first object within the three-dimensional object. If an internal object does not exist, the function returns NIL.

FUNCTION FInSymDef(h : Handle) : Handle;

Function FInSymDef receives a handle to a symbol definition in a symbol library and returns a handle to the first object within the symbol definition. If an internal object does not exist, the function returns NIL.

Layers and symbol folders are similar to grouped objects in that they both store internal objects. These internal objects can be accessed with the following routines:

FUNCTION FInLayer(h : Handle) : Handle;

Function FInLayer receives a handle to a layer and returns a handle to the first object within the layer. If an internal object does not exist, the function returns NIL.

FUNCTION FInFolder(h : Handle) : Handle;

Function FInFolder receives a handle to a symbol library folder and returns a handle to the first object within the folder. If an internal object does not exist, the function returns NIL.

The following routine may be used to receive a handle to an object according to its coordinate location on the screen.

FUNCTION PickObject(x,y : REAL) : Handle;

Function PickObject receives a coordinate location, specified by the parameters x, y. If the location is near a graphic object on the active layer, the function returns a handle to the graphic object. If there is not a graphic object near the location, the function returns NIL.

Section 7-3 **Handle Routines which Return Information from a Graphic Object**

Once the programmer has established a handle connection, what can he do with this handle? Well, he can obtain almost any type of information that the object stores. For example, the pre-defined function GetType returns the type of an object. If we continue to build upon the previous example,

Subprogram Example 7-3

```
PROCEDURE HandleEx;
VAR
  h : Handle;
  i : Integer;
BEGIN
  h := FObject;
  IF h <> NIL THEN i := GetType(h);
END;
RUN(HandleEx);
```


The numeric value of *i* represents the type of object that *h* is connected to (The possible results from Function *GetType* and their meaning will be discussed shortly). If *i* equals two, the programmer knows that he has established a connection to a line object. If the programmer knows that he is connected to a line object, then he can also get the line's endpoint locations. This would be achieved as follows:

Subprogram 7-4

```
PROCEDURE HandleEx;
VAR
  h : Handle;
  i : Integer;
  x1,y1,x2,y2 : Real;
BEGIN
  h := FObject;
  IF h <> NIL THEN
    BEGIN
      i := GetType(h);
      IF i = 2 THEN
        BEGIN
          GetSegPt1(h,x1,y1);
          GetSegPt2(h,x2,y2);
        END;
      END;
    END;
  END;
  RUN(HandleEx);
```

The procedures *GetSegPt1* and *GetSegPt2* return the endpoints of the line object to which *h* is connected. The point returned in *GetSegPt1* is the first point created by the user and the point returned in *GetSegPt2* is the last point created by the user. Many routines exist which allow the programmer to obtain information from a graphic object, layer or file. Some of the routines do not apply to all objects (e.g. *GetSegPt1* can only be used to get information from a line object). It is the responsibility of the programmer to ensure that the proper handle routines are used with the proper graphic objects.

Important Note:

It is the responsibility of the programmer to ensure that the proper handle routines are used with the proper graphic objects.

The following routines return information about graphic object. Remember, the programmer must check that the handle variables passed to these routines are valid.

FUNCTION GetType(h : Handle) : INTEGER;

Function GetType returns the type of the object that h is connected to. The object type values are as follows:

<u>Object</u>	<u>Value</u>	<u>Object</u>	<u>Value</u>
Line	2	Symbol Instance	15
Rectangle	3	Symbol Definition	16
Oval	4	Locus	17
Polygon	5	Spreadsheet	18
Arc	6	Extruded Object	24
FreeHand	8	Poly3D	25
Text	10	Layer	31
Group or Folder	11	Sweep Object	34
Quarter Arc	12	Mult. Extruded	38
Rounded Rectangle	13	Mesh	40

FUNCTION GetName(h : Handle) : STRING;

Function GetName returns the object name of the object to which h is connected. It returns None if the object has no name. (Note: A handle to a layer may not be passed to this routine).

FUNCTION GetClass(h : Handle) : STRING;

Function GetClass returns the class name of the object to which h is connected. It returns None if the object has no class name. (Note: A handle to a layer may not be passed to this routine).

PROCEDURE GetBBox(h : Handle; VAR x1,y1,x2,y2 : REAL);

Procedure GetBBox returns the bounding box of the object connected to h. The parameters x1 and y1 refer to the top-left corner of the object's bounding box while the parameters x2 and y2 refer to the bottom-right corner. (Note: Handles to loci and layers may not be used with this routine).

FUNCTION GetFPat(h : Handle) : INTEGER;









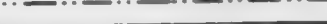
Function GetFPat returns the fill pattern of the object to which h is connected. (Note: The following objects may not be passed to this routine: locus, spreadsheet, line, layer, group, symbol instances and definitions, and three-dimensional objects). The numeric values returned correspond with those described with the routine FillPat.

FUNCTION GetLW(h : Handle) : INTEGER;

Function GetLW returns the line weight of the object to which h is connected. The value returned represents the width in mils. (Note: The following objects may not be passed to this routine: locus, spreadsheet, layer, group, symbol instances and definitions, and three-dimensional objects).

FUNCTION GetLS(h : Handle) : INTEGER;

Function GetLS returns the line style of the object to which h is connected. (Note: The following objects may not be passed to this routine: locus, spreadsheet, layer, group, symbol instances and definitions, and three-dimensional objects). The values returned represent the following line styles:

Line Styles	Value
	2
	-1
	-2
	-3
	-4
	-5
	-6
	-7
	-8

FUNCTION Selected(h : Handle) : BOOLEAN;

Function Selected returns whether an object is selected or not. If the object which is connected to h is selected, the function returns true, otherwise false.

FUNCTION GetText(h : Handle) : STRING;

If h is connected to a text object, then Function GetText returns the text contained within the object.

PROCEDURE GetSegPt1(h : Handle; VAR x,y : REAL);

If h is a handle to a line object, then Procedure GetSegPt1 returns the first endpoint of the line.

PROCEDURE GetSegPt2(h : Handle; VAR x,y : REAL);

If h is a handle to a line object, then Procedure GetSegPt2 returns the second endpoint of the line.

FUNCTION HasDim(h : Handle) : BOOLEAN;

If h is connected to a line object, then Function HasDim returns TRUE if the line has dimension text associated with it, otherwise it returns FALSE.

FUNCTION GetDimText(h : Handle) : STRING;

If h is connected to a dimensioned line or arc, then Function GetDimText returns the dimension value displayed with the object.

FUNCTION GetSymName(h : Handle) : STRING;

If h is connected to a symbol instance (a symbol in a drawing, not in a library), then Function GetSymName returns the symbol name of the instance.

FUNCTION GetSymRot(h : Handle) : REAL;

If h is a handle to a symbol instance, then Function GetSymRot returns the rotation value of the symbol.

PROCEDURE GetLocPt(h : Handle; VAR x, y : REAL);

If h is a handle to a locus, then Procedure GetLocPt returns the coordinate location of the locus in its x and y variable parameters.

PROCEDURE GetRRDiam(h : Handle;

VAR hDiameter, vDiameter: REAL);

Procedure GetRRDiam, if passed a handle to a rounded rectangle, returns the horizontal and vertical diameters of its rounded corners.

**PROCEDURE GetArc(h : Handle;
VAR startAngle,arcAngle : REAL);**

Procedure GetArc, if passed a handle to an arc, returns the start and arc angle of the arc.

FUNCTION GetVertNum(h : Handle) : INTEGER;

If h is a handle to a polygon, then Function GetVertNum returns the number of vertices contained within the object.

**PROCEDURE GetPolyPt(h : Handle;
index : INTEGER; VAR x, y : REAL) : INTEGER;**

If h is a handle to a polygon and the value of index is less than or equal to the number of vertices in the polygon, then Procedure GetPolyPt returns the point of the vertex specified by the index. (Note: the first vertex of a polygon has an index of 1).

**PROCEDURE Get3DCntr(h : Handle;
VAR x,y,z : REAL);**

If h is a handle to a three-dimensional object, then Procedure Get3DCntr returns the three-dimensional center point of the object.

**PROCEDURE Get3DInfo(h : Handle;
VAR height, width, depth : REAL);**

If h is a handle to a three-dimensional object, then Procedure Get3DInfo returns the height, width and depth values of the object in its variable parameters.

**PROCEDURE SprdSize(h : Handle;
VAR rows,columns : INTEGER);**

If h is a handle to a spreadsheet, then Procedure SprdSize returns the number of rows and columns within it.

**FUNCTION CellHasStr(h : Handle;
row,column : INTEGER) : BOOLEAN;**

If h is a handle to a spreadsheet, then Function CellHasStr returns TRUE if the specified cell contains a string. The function returns FALSE if the cell does not

contain a string. (Note: a cell has a string even if it is an equation).

**FUNCTION CellHasNum(h : Handle;
row,column : INTEGER) : BOOLEAN;**

If h is a handle to a spreadsheet, then Function CellHasNum returns TRUE if the specified cell contains an equation which returns a numeric value. The function returns FALSE if the cell does not contain such an equation.

**FUNCTION GetCellStr(h : Handle;
row,column : INTEGER) : STRING;**

If h is a handle to a spreadsheet, then Function GetCellStr returns the string located in the specified cell.

**FUNCTION GetCellNum(h : Handle;
row,column : INTEGER) : REAL;**

If h is a handle to a spreadsheet, then Function GetCellNum returns the numeric value located in the specified cell.

**FUNCTION GetCWidth(h : Handle;
row,column : INTEGER) : INTEGER;**

If h is a handle to a spreadsheet, then Function GetCWidth returns the column width of the cell specified.

**FUNCTION GetCAlign(h : Handle;
row,column : INTEGER) : INTEGER;**

If h is a handle to a spreadsheet, then Function GetCAlign returns the alignment value of the cell specified. The function results correspond to the following alignment values:

<u>Alignment Value</u>	<u>Setting</u>
1	General
2	Left
3	Right
4	Center

FUNCTION HArea(h : Handle) : REAL;

If h is a handle variable connected to a rectangle, rounded-rectangle, arc, group, quarter-arc, oval, polygon, freehand-polygon, extruded object or symbol instance, then Function HArea returns the area of the object.

FUNCTION HPerim (h : Handle) : REAL;

If h is a handle variable connected to a rectangle, rounded-rectangle, arc, group, quarter-arc, oval, polygon, freehand-polygon, extruded object or symbol instance, then Function HPerim returns the perimeter of the object.

FUNCTION HLength (h : Handle) : REAL;

If h is a handle to a line, then Function HLength returns the length of the line.

FUNCTION HWidth (h : Handle) : REAL;

If h is a handle to a graphic object other than a locus, then Function HWidth returns the width of the object.

FUNCTION HHeight (h : Handle) : REAL;

If h is a handle to a graphic object other than a locus, then Function HHeight returns the height of the object.

FUNCTION HAngle (h : Handle) : REAL;

If h is a handle to a line or an arc, then Function HAngle returns the angle of the object.

PROCEDURE HCenter (h: Handle; VAR x,y : REAL);

If h is a handle to a graphic object other than a locus, then Procedure HCenter returns the center point of the object.

Section 7-4 **Handle Routines which Return Information from a Layer**

The following routines return information from a handle variable connected to a layer:

FUNCTION GetLName(h : Handle) : STRING;

If h is a handle connected to a layer, then Function GetLName returns the name of the layer.

FUNCTION NumObj(h : Handle) : INTEGER;

If h is a handle connected to a layer, then Function NumObj returns the number of objects on the layer.

FUNCTION NumSObj(h : Handle) : INTEGER;

If h is a handle connected to a layer, then Function NumSObj returns the number of selected objects on the layer.

FUNCTION GetLScale(h : Handle) : REAL;

If h is a handle connected to a layer, then Function GetLScale returns the scale of the layer.

Section 7-5 **Handle Routines which Return Information from a Symbol Library.**

The following routines return information from a handle variable connected to a symbol definition:

FUNCTION GetSDName(h : Handle) : STRING;

If h is a handle to a symbol definition, then Function GetSDName returns the name of the symbol definition.

Section 7-6 Handle Routines which Return Information from a Drawing

The following routines return information from a Mini-Cad+ drawing.

FUNCTION GetFName : STRING;

Function GetFName returns the current file name of the active drawing.

FUNCTION NumLayers: INTEGER;

Function NumLayers returns the current number of layers within the active drawing.

FUNCTION SymDefNum : INTEGER;

Function SymDefNum returns the number of symbol definitions within the active drawing.

FUNCTION NameNum : INTEGER;

Function NameNum returns the number different object names in the active drawing.

FUNCTION NameList(index : INTEGER) : STRING;

Function NameList returns the object name stored in the object name list designated by the value of index. (Note: the first object name index is 1).

FUNCTION ClassNum : INTEGER;

Function ClassNum returns the number of different class names in the active drawing.

FUNCTION ClassList(index : INTEGER) : STRING;

Function ClassList returns the class name stored in the class name list designated by the value of index. (Note: the first class name index is 1).

PROCEDURE GetOrigin(VAR x,y : REAL);

Procedure GetOrigin returns the current origin location relative to the center of the page.

FUNCTION FArrowHead : INTEGER;

Function FArrowHead returns the current arrowhead setting. The result of FArrowHead corresponds to those described with the routine ArrowHead.

FUNCTION FFillPat : INTEGER;

Function FFillPat returns the current fill pattern setting. The result of FFillPat corresponds to those described with the routine FillPat.

FUNCTION FPenSize : INTEGER;

Function FPenSize returns, in mils, the current pen size setting.

FUNCTION FPenPat : INTEGER;

Function FPenPat returns the current pen pattern setting. The result of FPenPat corresponds to those described with the routine PenPat.

PROCEDURE FPenFore(VAR red, green, blue : LONGINT);

Procedure FPenFore returns the current pen foreground color in the variable parameters red, green and blue.

PROCEDURE FPenBack(VAR red, green, blue : LONGINT);

Procedure FPenBack returns the current pen background color in the variable parameters red, green and blue.

PROCEDURE FFillFore(VAR red, green, blue : LONGINT);

Procedure FFillFore returns the current fill foreground color in the variable parameters red, green and blue.

PROCEDURE FFillBack(VAR red, green, blue : LONGINT);

Procedure FFillBack returns the current fill background color in the variable parameters red, green and blue.

Section 7-7 Handle Routines which Change a Graphic Object

The following is a list of handle routines which change a graphic object:

PROCEDURE SetSelect(h : Handle);

If h is a handle to a graphic object, then Procedure SetSelect selects the object.

PROCEDURE SetDSelect(h : Handle);

If h is a handle to a graphic object, then Procedure SetDSelect deselects the object.

PROCEDURE SetName(h : Handle; name : STRING);

If h is a handle to a graphic object, then Procedure SetName assigns the specified name to the object.

PROCEDURE SetClass(h : Handle; class : STRING);

If h is a handle to a graphic object, then Procedure SetClass assigns the specified class to the object.

PROCEDURE SetFPat(h : Handle; fillpattern : INTEGER);

If h is a handle to a graphic object which can store a fill pattern, then Procedure SetFPat assigns the specified fill pattern to the object. Refer to the routine FillPat for the available fill pattern values.

PROCEDURE SetLW(h : Handle; linewidth : INTEGER);

If h is a handle to a graphic object which has a line weight, then SetLW assigns the specified line weight (in mils) to the object.

PROCEDURE SetLS(h : Handle; linestyle : INTEGER);

If h is a handle to a graphic object which has a line style, then SetLS assigns the specified line style to the object. If the parameter linestyle contains a value in the range [0..71], then Procedure SetLS changes the pen pattern of the graphic object, otherwise the line style is changed.

PROCEDURE SetBBox(h : Handle; x1,y1, x2, y2 : REAL);

If h is a handle to a graphic object which is not a line or locus, then Procedure SetBBox changes the bounding box of the graphic object. The parameters x1, y1 refer to the upper-left bounding box corner while the parameters x2, y2 refer to the bottom-right bounding box corner.

**PROCEDURE SetPenFore(h : Handle;
red, green, blue : LONGINT);**

If h is a handle to a graphic object, then Procedure SetPenFore changes the pen foreground color setting of the object.

**PROCEDURE SetPenBack(h : Handle;
red, green, blue : LONGINT);**

If h is a handle to a graphic object, then Procedure SetPenBack changes the pen background color setting of the object.

**PROCEDURE SetFillFore(h : Handle;
red, green, blue : LONGINT);**

If h is a handle to a graphic object, then Procedure SetFillFore changes the fill foreground color setting of the object.

**PROCEDURE SetFillBack(h : Handle;
red, green, blue : LONGINT);**

If h is a handle to a graphic object, then Procedure SetFillBack changes the fill background color setting of the object.

PROCEDURE SetSegPt1(h : Handle; x,y : REAL);

If h is a handle to a line, then Procedure SetSegPt1 changes the location of the first endpoint to the coordinate provided.

PROCEDURE SetSegPt2(h : Handle; x,y : REAL);

If h is a handle to a line, then Procedure SetSegPt2 changes the location of the second endpoint to the coordinate provided.

**PROCEDURE SetPolyPt(h : Handle; index : INTEGER;
x,y : REAL);**

If h is a handle to a polygon, then Procedure SetPolyPt changes the location of the polygon's vertex, which is specified by the parameter index, to the coordinate provided.

**PROCEDURE SetArc(h : Handle;
#startAngle, #arcAngle : REAL);**

If h is a handle to an arc, then Procedure SetArc changes the arc's startAngle and arcAngle values.

**PROCEDURE Set3DRot(h : Handle; #x, #y, #z,
xCenter,yCenter,zCenter : REAL);**

If h is a handle to a three-dimensional object, then Procedure Set3DRot rotates the object by the x, y and z angle values provided around the three-dimensional point specified by the parameters xCenter, yCenter and zCenter.

**PROCEDURE Set3DInfo(h : Handle; height, width,
depth : REAL);**

If h is a handle to a three-dimensional object, then Procedure Set3DInfo sets the height, width and depth dimensions of the object.

PROCEDURE Move3DObj(h : Handle; x,y,z : REAL);

If h is a handle to a three-dimensional object, then Procedure Move3DObj offsets the object by the distance values specified in the parameters x, y and z.

PROCEDURE SelectSS(h : Handle);

If h is a handle to a spreadsheet, then Procedure SelectSS activates the spreadsheet.

PROCEDURE SetText(h : Handle; s : STRING);

If h is a handle to a text object, then Procedure SetText changes the object's text to the text specified by the parameter s. (Note: the text object retains its font, size and style characteristics).

Section 7-8 **Handle Routines which Change File Settings**

The following routines change file settings:

**PROCEDURE SetView(h : Handle; #x, #y, #z,
 xCenter,yCenter,zCenter : REAL);**

Procedure SetView rotates the file's view by the x, y and z angle values provided around the three-dimensional point specified by the parameters xCenter, yCenter and zCenter.

PROCEDURE SetCursor(cursorType);

Procedure SetCursor changes the cursor's appearance. The possible values for the parameter cursorType include:

<u>Cursor</u>	<u>CursorType Parameter</u>
Large cross	LgCrossC
Small cross	SmCrossC
Watch	WatchC
Text Bar	TextBarC
Arrow	ArrowC
Hand	HandC

Section 7-9 Handle Traversing Routines

Procedure For Each Object

If the user is storing a handle to a layer, he may move to the next layer in the following manner.

Subprogram Example 7-5

```
PROCEDURE HandleEx;  
VAR  
    h : Handle;  
BEGIN  
    h := FLayer;  
    h := NextLayer(h);  
END;  
RUN(HandleEx);
```

The routine NextLayer connects the handle variable h to the next layer relative to h's current layer location. Therefore, if h is connected to the first layer, the routine NextLayer connects the variable h to the second layer (assuming there is a second layer). Thus, the layers form a list which the programmer can traverse. However, every list has an end. The programmer cannot tell when he is going to reach the end of a list unless he makes a check for it. The end of every list returns the value NIL to the handle variable. Therefore, if there was no second layer in the above subprogram example, the value of h would be NIL after the NextLayer call. The example below shows a check for a NIL handle.

Subprogram Example 7-6

```
PROCEDURE HandleEx;  
VAR  
    h : Handle;  
    s : String;  
BEGIN  
    h := FLayer;  
    h := NextLayer(h);  
    IF h <> NIL THEN s := GetLName(h);  
END;  
RUN(HandleEx);
```

The following routines allow the programmer to traverse the layer list:

FUNCTION NextLayer(h : Handle) : Handle;

If h is a handle to a layer, then Function NextLayer returns a handle to the next layer in the list. If a next layer does not exist, then the function returns NIL.

FUNCTION PrevLayer(h : Handle) : Handle;

If h is a handle to a layer, then Function PrevLayer returns a handle to the previous layer in the list. If there is not a previous layer, then the function returns NIL.

If a programmer has a handle variable to a graphic object, he may traverse the graphic object list by using the following routines:

FUNCTION NextObj(h : Handle) : Handle;

If h is a handle to a graphic object, then Function NextObj returns the next object in the list. If there is not a next object, the function returns NIL.

FUNCTION PrevObj(h : Handle) : Handle;

If h is a handle to a graphic object, then Function PrevObj returns the previous object in the list. If there is not a previous object, the function returns NIL.

FUNCTION NextSObj(h : Handle) : Handle;

If h is a handle to a graphic object, then Function NextSObj returns a handle to the next selected object in the list. If no other object in the remaining list is selected, then the function returns NIL.

FUNCTION PrevSObj(h : Handle) : Handle;

If h is a handle to a graphic object, then Function PrevSObj returns the previous selected object in the list. If there is no object which is selected prior to the specified object, the function returns NIL.

FUNCTION NextDObj(h : Handle) : Handle;

If h is a handle to a graphic object, then Function NextDObj returns the next deselected object in the list. If no other object in the remaining list is deselected, then the function returns NIL.

FUNCTION PrevDObj(h : Handle) : Handle;

If h is a handle to a graphic object, then Function PrevDObj returns the previous deselected object in the list. If there is no object which is deselected prior to the specified object, the function returns NIL.

As you can see, if the programmer has a handle variable to an object or a layer, he can traverse through the layer or object list to gain access to other layers or objects. The programmer can do the same thing with the symbol library. The routines for traversing the symbol library are as follows:

FUNCTION NextSymDef(h : Handle) : Handle;

If h is a handle to a symbol definition, then Function NextSymDef returns a handle to the next symbol definition in the library. If no symbol definition exists after the one specified, the function returns NIL.

FUNCTION PrevSymDef(h : Handle) : Handle;

If h is a handle to a symbol definition, then Function PrevSymDef returns a handle to the previous symbol definition in the library. If no symbol definition exists prior to the one specified, the function returns NIL.

Unit VIII Using Array and Vector Variables.

Introduction This unit discusses the use of *Array* and *Vector* variables within the MiniPascal programming language.

Section 8-1 Array Variables

What are *Array* variables? *Array* variables are a means of storing information. They allow the programmer to store elements of data in a pre-defined location which can be easily accessed at a later time. An *Array* is a data type which is declared as follows:

Subprogram Example 8-1

```
PROCEDURE ArrayEx;
VAR
  i : INTEGER;
  anArray : ARRAY [1..10] OF REAL;
BEGIN
  FOR i := 1 TO 10 DO
    BEGIN
      anArray[i] := i * 10;
    END;
  FOR i := 1 TO 10 DO
    BEGIN
      Writeln('anArray[' , i , ']' = ' , anArray[i]:10:0);
    END;
  END;
RUN(ArrayEx);
```

An array variable declaration specifies three things:

1. The type of information the array stores.
2. The number of elements the array can hold.
3. The method of referring to the array's elements.

Specifying the type of information that the array can store is simple. An array can store any of the following data types: *Integer*, *Longint*, *Real*, *Char*, *String*, *Handle*, or *Boolean*. The data type is specified after the reserved word *OF* in the declaration statement.

The number of elements that an array can hold is dependent upon an array's size. The size of an array is specified by the lower and upper bounds of each array dimension. An array dimension is specified by its limit values - the array bounds - which must be defined by

numerals between -32768 and 32767. The bounds are separated by two dots (..) which have the meaning "through and including." If an array has two dimensions, the pairs of array bounds are separated by commas. Subprogram Example 8-2 demonstrates an array with two dimensions:

Subprogram Example 8-2

```
PROCEDURE ArrayEx;
VAR
  i,j : INTEGER;
  anArray : ARRAY [1..10,1..2] OF REAL;
BEGIN
  FOR i := 1 TO 10 DO
    BEGIN
      FOR j := 1 TO 2 DO
        anArray[i,j] := i * j;
      END;
    FOR i := 1 TO 10 DO
      BEGIN
        FOR j := 1 TO 2 DO
          Writeln('anArray[' ,i ,',' ,j ,'] = ',anArray[i,j]:10:0);
        END;
      END;
    END;
  RUN(ArrayEx);
```

Finally, the values used as array bounds indicate how array elements can be accessed. Individual elements are stored in and retrieved from Array variables according to their location within the array. The location is given by a subscript, a name which comes from the arrays used in physics and mathematics. In MiniPascal, an *array* variable's subscript is given between brackets. To access a single dimension array element, place the element's subscript immediately after the array variable. If more than one subscript is required (e.g. arrays which have two dimensions), separate the subscripts with commas. Also, the programmer can use variables or expressions to specify subscripts.

Array variables can be used for a wide range of programming projects. A basic use is for sorting data. A very obvious sorting routine known as a selection sort works as follows:

Suppose that we have an array of *integer* values:

32	25	77	83	19	4	43	62	99
----	----	----	----	----	---	----	----	----

Search through the array (starting at the first array element), find the smallest value, and exchange it with the value stored in the first array location.

4	25	77	83	19	32	43	62	99
---	----	----	----	----	----	----	----	----

Next, search through the array again (starting at the second array element), find the smallest value, and exchange it with the value stored in the second array location.

4	19	77	83	25	32	43	62	32
---	----	----	----	----	----	----	----	----

Continue the exchange process until we reach the end of the array.

This sorting process can be simulated using arrays.

Subprogram Example 8-3

PROCEDURE Sort;

CONST

arrayLimit = 9;

VAR

theArray : ARRAY[1..9] OF INTEGER;

first,current,smallest : INTEGER;

PROCEDURE Switch(index1,index2 : INTEGER);

VAR

temp : INTEGER;

BEGIN

temp := theArray[index1];

theArray[index1] := theArray[index2];

theArray[index2] := temp;

END;

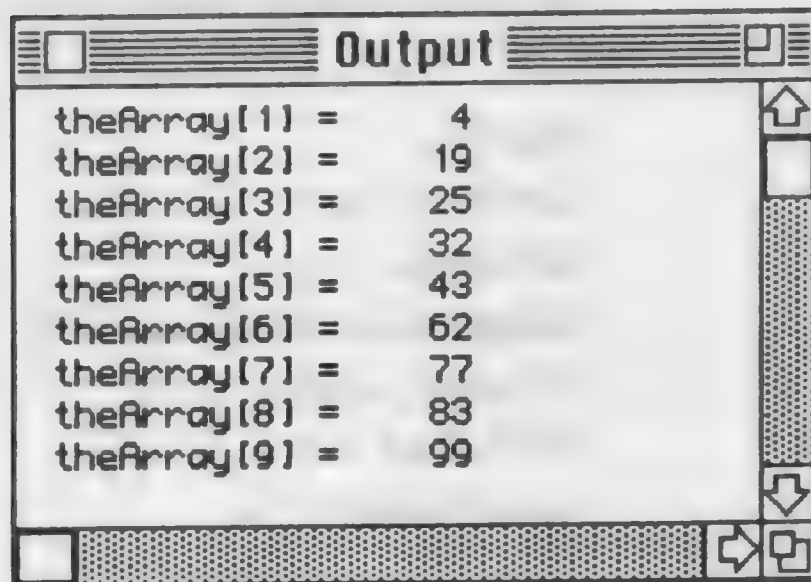
BEGIN

```
theArray[1] := 32;
theArray[2] := 25;
theArray[3] := 77;
theArray[4] := 83;
theArray[5] := 19;
theArray[6] := 4;
theArray[7] := 43;
theArray[8] := 62;
theArray[9] := 99;
FOR first := 1 TO (arrayLimit - 1) DO
  BEGIN
    smallest := first;
    FOR current := first TO arrayLimit DO
      IF theArray[current] < theArray[smallest]
      THEN smallest := current;
    Switch(smallest,first);
  END;
FOR current := 1 TO arrayLimit DO
  Writeln('theArray[' ,current, ']' = ' ,theArray[current]:5);
END;
RUN(Sort);
```

The output from Subprogram Example 8-3 is displayed in Diagram 8-1:

Diagram 8-1

Output from Subprogram Example 8-3



Section 8-2 Vector Variables

What are *vectors*? A *vector* is a mathematical tool for calculating a physical quantity which has a magnitude and a direction. Such quantities include velocity, acceleration, force, etc. A *vector* quantity is represented by a directed line segment, the length of which represents the magnitude of the *vector*. In MiniPascal, a *vector* variable is declared as follows:

```
VAR
    aVector : Vector;
```

A *vector* stores three values of type *real*. Each value represents a location along the x, y and z axis. A value within a *vector* may be stored or retrieved through the use of an index in the range 1..3. For Example:

Subprogram Example 8-4

```
PROCEDURE Test;
VAR
    aVector : Vector;
BEGIN
    aVector[1] := 1;
    aVector[2] := 1;
    aVector[3] := 0;
    Writeln('aVector[1] = ', aVector[1]);
    Writeln('aVector[2] = ', aVector[2]);
    Writeln('aVector[3] = ', aVector[3]);
END;
RUN(Test);
```

The following operations may be performed upon vectors (*v* and *w* being vectors and *k* a nonzero real number):

Negative: $-v$

Addition: $v + w$

Subtraction: $v - w$

Multiplication with a scalar: $k * v$

Division by a scalar: v / k

Dot Product: $v \bullet w$

Cross Product: $v * w$

The following is a list of pre-defined vector routines:

FUNCTION Norm(v : Vector) : REAL;

Function Norm returns the norm (or length) of the vector v.

FUNCTION Perp(v : Vector) : Vector;

Function Perp returns a vector which is perpendicular to vector v.

FUNCTION Vec2Ang(v : Vector) : REAL;

Function Vec2Ang returns the angle of vector v. The angle result will be in the range $-180 < \text{Vec2Ang} \leq 180$.

FUNCTION Ang2Vec(#angle, length : REAL) : REAL;

Function Ang2Vec receives an angle and a length and returns a vector.

FUNCTION UnitVec(v : Vector) : Vector;

Function UnitVec returns a standard unit vector of v.

PROCEDURE Comp(u, a : Vector; VAR w1, w2 : Vector);

Procedure Comp receives two vectors, u and a, and returns the vector component of u along a in w1 and the vector component of u orthogonal to a in w2.

FUNCTION AngBVec(u, a : Vector) : REAL;

Function AngBVec receives two vectors and returns the angle (in radians) between the two vectors. The angle returned will be in the range $-\pi < \text{AngBVec} \leq \pi$.

Unit IX Interactive Routines.

Introduction This unit discusses the use of interactive routines within the MiniCad+ environment. Interactive routines allow the programmer to actually look for specific activities (events) performed by the user and then respond to these activities. These routines will extend the programmer's control by allowing him to monitor actions which occur by the mouse or the keyboard.

Section 9-1 Routines which Continue to Execute Until a Specific Event Occurs.

Several interactive routines continue to execute until the user responds with the proper event. For example, if the programmer calls Procedure GetPt, the routine will continue to execute until the user presses the mouse button within the active MiniCad+ drawing window.

PROCEDURE GetRect(VAR x1,y1,x2,y1 : REAL);

When Procedure GetRect is executed, the palette tool automatically changes to the rectangle tool. The routine continues to execute until the user creates a rectangle with the mouse. As the user creates the rectangle, a black border appears while it is being drawn. Also, the rectangle can be constrained using the constrain boxes at the bottom of the document window. The top-left and bottom-right corners of the rectangle are returned in the variable parameters x1, y1, x2, y2.

Subprogram Example 9-1

```
PROCEDURE ATest;  
VAR  
    x1,y1,x2,y2 : REAL;  
BEGIN  
    GetRect(x1,y1,x2,y2);  
    Oval(x1,y1,x2,y2);  
END;  
RUN( ATest );
```

PROCEDURE GetLine(VAR x1,y1,x2,y1 : REAL);

When Procedure GetLine is executed, the palette tool automatically changes to the line tool. The routine con-

tinues to execute until the user creates a line with the mouse. As the user creates the line, a black line follows the mouse as it is drawn. Also, the line may be constrained using the constrain boxes located at the bottom of the MiniCad+ document. When the line is drawn, the first and second endpoints of the line are returned in the variable parameters $x1, y1, x2, y2$.

Subprogram Example 9-2

```
PROCEDURE ATest;
VAR
    x1,y1,x2,y2 : REAL;
BEGIN
    GetLine(x1,y1,x2,y2);
    MoveTo(x1,y1);
    LineTo(x2,y2);
    DimText();
END;
RUN(ATest);
```

PROCEDURE GetPt(VAR X,Y : REAL);

When Procedure GetPt is called, the routine continues to execute until the user clicks the cursor within the active MiniCad+ drawing window. When the mouse button is depressed, the location of the mouse is returned in the variable parameters X and Y.

Subprogram Example 9-3

```
PROCEDURE ATest;
VAR
    x,y : REAL;
BEGIN
    GetPt(x,y);
    Locus(x,y);
END;
RUN(ATest);
```

PROCEDURE GetPtL(X1,Y1 : REAL; VAR X2,Y2 : REAL);

When Procedure GetPtL is called, a line with its first endpoint at the coordinates $X1, Y1$ appears. The second endpoint continues to follow the mouse until the user clicks down. The user may constrain the line with the constrain boxes located at the bottom of the MiniCad+ document. The location of second endpoint is returned in the variable parameters $X2, Y2$.

Subprogram Example 9-4

```
PROCEDURE ATest;  
VAR  
    x,y : REAL;  
BEGIN  
    GetPtL(0,0,x,y);  
    Locus(x,y);  
END;  
RUN(ATest);
```

PROCEDURE GetKeyDown(VAR aCode:INTEGER);

When Procedure GetKeyDown is called, the routine continues to execute until the user depresses a key on the keyboard. When a key is depressed, the ASCII code of the keyboard character is returned in the variable parameter aCode.

Subprogram Example 9-5

```
PROCEDURE ATest;  
VAR  
    aCode : INTEGER;  
    s : STRING;  
BEGIN  
    GetKeyDown(aCode);  
    s := Chr(aCode);  
    TextOrigin(0,0);  
    BeginText;  
        s  
    EndText;  
END;  
RUN(ATest);
```

Section 9-2 Routines which Monitor for User Activities.

The following routines are executed within a REPEAT..UNTIL or a WHILE..DO loop. Their use is to notify the programmer when an event (a mouse down or a key down) has occurred. These routines differ from the previous interactive routines in that they allow the programmer to do other things rather than just wait until the user performs a specific activity. The routines return TRUE if the specified event has occurred.

FUNCTION MouseDown(VAR x,y : REAL) : BOOLEAN;

Function MouseDown returns TRUE if a mouse down event has occurred within the active MiniCad+ document window, otherwise it returns FALSE. If a mouse down event has occurred, the variable parameters x and y, of type *Real*, return the location of the mouse in MiniCad+ coordinates.

Subprogram Example 9-6

PROCEDURE ATest;

VAR

 x1,y1,x2,y2 : REAL;

BEGIN

REPEAT

UNTIL MouseDown(x1,y1);

REPEAT

UNTIL MouseDown(x2,y2);

 MoveTo(x1,y1);

 LineTo(x2,y2);

IF Option **THEN** DimText();

END;

RUN(ATest);

**FUNCTION KeyDown(VAR asciiCode : INTEGER) :
 BOOLEAN;**

Function KeyDown returns TRUE if a non-modifier keyboard character has been depressed, otherwise it returns FALSE. If a keyboard character has been depressed then the variable parameter asciiCode returns the ASCII code of the character. For the example below, create a 3 dimensional object, select it and then run the macro. The user can rotate the object by using the '+' and '-' keys. Click the mouse to stop the macro.

Subprogram Example 9-7

```
PROCEDURE Test;
LABEL 1;
VAR
    h : Handle;
    xC,yC,zC,xR,yR,zR,x,y : REAL;
    cCode : INTEGER;
BEGIN
    h := FSActLayer;
    IF h = NIL THEN GOTO 1;
    Get3DCntr(h,xC,yC,zC);
    AngDialog3('Enter the rotation increment : ', '0°','0°','0°',
        xR,yR,zR);
    WHILE NOT MouseDown(x,y) DO
        BEGIN
            IF KeyDown(cCode) | AutoKey(cCode) THEN
                BEGIN
                    IF cCode = 43
                    THEN
                        Set3DRot(h,xR,yR,zR,xC,yC,zC)
                    ELSE IF cCode = 45
                    THEN
                        Set3DRot(h,-xR,-yR,-zR,xC,yC,zC);
                    Redraw;
                END;
            END;
        END;
    1:END;
RUN(Test);
```

```
FUNCTION AutoKey(VAR asciiCode : INTEGER) :
    BOOLEAN;
```

Function AutoKey returns TRUE if a non-modifier keyboard character has been continually depressed, otherwise it returns FALSE. If a keyboard character has been continually depressed, then the variable parameter asciiCode returns the ASCII code of the character. (Note: an autokey event occurs when a key has been depressed long enough for a key repeat to occur. The amount of time for this event to occur is relative to the key delay settings found in the Control Panel.)

PROCEDURE GetMouse(VAR x,y : REAL);

Procedure GetMouse returns the current location of the cursor while it is within an active MiniCad+ document. (Note: Procedure GetMouse differs from Procedure GetPt in that Procedure GetMouse returns the location of the cursor while the user is moving it around the screen, rather than waiting for the user to click the mouse.)

Subprogram Example 9-8

PROCEDURE ATest;

VAR

aCode : INTEGER;

x,y : REAL;

s : STRING;

BEGIN

WHILE NOT KeyDown(aCode) DO

BEGIN

GetMouse(x,y);

s:=Concat('X:',Num2StrF(x),'Y:',Num2StrF(y));

Message(s);

END;

END;

RUN(ATest);

FUNCTION Option : BOOLEAN;

FUNCTION CapsLock : BOOLEAN;

FUNCTION Shift : BOOLEAN;

FUNCTION Command : BOOLEAN;

These functions return TRUE if their corresponding modifier key was depressed during the last event by the user. (See Function MouseDown for an example).

These functions only apply to the following functions: Function MouseDown, Function KeyDown and Function AutoKey.

Section 9-3 Routines for Creation of Custom Dialog .

Procedure BeginDialog

(dialogID, dialogType, x1, y1, x2, y2 : Integer);

Procedure BeginDialog initiates the dialog-creation process.

The parameter dialogID of type Integer, assigns a unique ID number to the dialog being created. The parameter dialogID must be in the range [1..20]. The programmer will use this to refer to this dialog in other routines.

The parameter dialogType, of type Integer, specifies the type of dialog to be created. The parameter dialogType has a value in the range [1..3].

A value of 1 creates a standard dialog box.

A value of 2 creates a plain dialog box with no border.

A value of 3 creates a dialog box with a two-pixel thick shadow instead of a border

The parameters x1, y1, x2, y2, of type Integer, specify the top-left and bottom right corners of the dialog box. Note that these values are integer coordinates which refer to the current screen opposed to MiniCad+'s coordinate system. The screen coordinate location (0,0) is located at the top-left corner of the screen with positive x and y values moving towards the bottom-right corner. The procedure GetScreen (described later) provides the top-left and bottom-right coordinates of the current screen. Programmers should use this routine to insure that all of their dialog boxes will be centered properly regardless of the user's screen size.

Procedure EndDialog;

Procedure End Dialog terminates the dialog-creation Process.

Procedure AddButton (buttonStr : STRING; itemID, buttonType, x1, y1, x2, y2: Integer);

Procedure AddButton creates a button within the dialog currently being created (Procedure AddButton must be located within the BeginDialog... EndDialog calls).

The parameter buttonStr, of type STRING, specifies the text which will be placed within the button being created.

The parameter itemID, of type Integer, assigns a unique ID number to the button. The parameter itemID must be in the range [1..50].

Important Note:

Dialog may contain a combination of buttons and text fields. Each button or text field is considered a single item. Each item must have its own unique ID. Thus, a dialog with two buttons and two text fields would have to assign a different itemID to each item.

The parameter buttonType, of type Integer, specifies the type of button to be created. The parameter buttonType has a value in the range [1..3]. If buttonType has a value of 1, a standard button is created.



If buttonType has a value of 2, a check box button is created.



If buttonType has a value of 3, a radio button is created.



Important Note:

If the programmer creates a standard button (`buttonType=1`) and he assigns the value 1 as its `itemID` (`itemID=1`), this button will appear outlined and treated as the default button. A default button is automatically depressed if the user presses the Return key while the dialog is displayed. Below is an outlined button:



The parameters `x1`, `y1`, `x2`, `y2`, of type Integer, specify the top-left and bottom-right corners of the button. These coordinates are relative to the button's dialog box. Thus, the coordinate location (0,0) is the top left corner of the button's dialog box.

Procedure AddField (`fieldStr` : STRING; `itemID`, `fieldType`, `x1`, `y1`, `x2`, `y2` : INTEGER)

Procedure AddField creates a static or editable text field within the dialog currently being created.

The parameter `fieldStr`, of type STRING, specifies the text to be displayed with the text field.

The parameter `itemID`, of type Integer, specifies the unique ID which will be assigned to the `etxt` field. The parameter `itemID` has a value in the range [1..50].

The parameter `fieldType`, of type Integer, specifies the type of field to be created. If `fieldType` has a value of 1, a static text field is created. A value of 2 creates an editable text field,

The parameters `x1`, `y1`, `x2`, `y2`, of type Integer, specify the top-left and bottom-right corners of the text field.

Procedure GetDialog (`dialogId` : INTEGER);

Procedure GetDialog displays the dialog which has the ID specified by the parameter `dialogID`.

Procedure ClrDialog;

Procedure ClrDialog closes a dialog which is currently displayed.

Procedure DialogEvent (VAR item : INTEGER);

Procedure DialogEvent monitors user activity within a dialog. If the user has selected a button, selected an editable text field or depressed the Tab key when editable text field items exist in the dialog, then Procedure DialogEvent returns the ID number of the item selected in the variable parameter item, of type INTEGER.

Function GetField (fieldID: INTEGER) : STRING;

Function GetField returns the text located in the field which has the ID specified by the parameter fieldID.

Procedure SetField (fieldID: INTEGER ; str : STRING);

Procedure SetField places the text provided in the parameter str, of type STRING, into the text field which has the ID specified by the parameter fieldID. This routine may only be called while a dialog is displayed.

Procedure SelField (fieldID: INTEGER);

Procedure SelField hilites the text filed which has the ID specified by the parameter fieldID.

Procedure SetItem (fieldID: INTEGER; select : BOOLEAN);

Procedure SetItem selects or deselects the check box or radio button which has the ID specified by the parameter fieldID. If the parameter select, of type Boolean, is True, the button is selected, otherwise it is deselected.

Function ItemSel (fieldID : INTEGER) BOOLEAN;

Function ItemSel returns TRUE if the checkbox or radio button which is specified by the parameter fieldID is selected, otherwise it returns FALSE.

Function ValidNumStr (str: STRING; VAR value: Real) : BOOLEAN;

Function ValidNumStr returns TRUE if the string parameter str can be converted into a numeric value, otherwise it returns FALSE. If ValidNumStr does return TRUE, the value of the string is returned in the variable parameter value, of type Real.

Function ValidAngStr (str: STRING; VAR value: Real) : BOOLEAN;

Function ValidAngStr returns TRUE if the string parameter str can be converted into a numeric value, otherwise it returns FALSE. If ValidAngStr does return TRUE, the value of the string is returned (in decimal degrees) in the variable parameter value, of type Real.

Procedure GetScreen (VAR x1, y1, x2, y2: INTEGER);

Procedure GetScreen returns the top-left and bottom-right corners of the screen. These values will change as different sized screen are used.

Unit X

10.1 Database & Worksheet Routines.

**Procedure NEWFIELD(Format , FieldName ,
FieldValue : STRING; NumericForm, Accuracy : IN-
TEGER);**

Assigns a new field to a Format. Creates a new Format if one does not exist by the name listed in the Procedure string.

**Procedure SETRECORD(ObjectHandle : HANDLE;
TheFormat : STRING);**

Assigns a record to the object that is specified by the ObjectHandle and the type of record is specified by TheFormat.

**Procedure SETFIELD(ObjectHandle : HANDLE; For-
mat , Field , Value : STRING);**

Assigns a field a value of a previously created record assigned to an object.

**Function EVAL(ObjectHandle : HANDLE; Search-
Criteria : STRING) : REAL;**

Given a SearchCriteria for example 'count = rect'; count number of rectangles in layer, EVAL will return a number of type REAL. If the SearchCriteria is of type BOOLEAN then EVAL will return 0.0 or 1.0;

FALSE = 0.0

TRUE = 1.0

**Function EVALSTR(Object Handle:HANDLE,
SearchCriteria : STRING) : STRING;**

Given a SearchCriteria for example 'count = rect'; count number of rectangles in layer, EVALSTR will return a number of type STRING. If the SearchCriteria returns a type REAL then EVALSTR will convert it to type STRING. This mean that the value returned by EVALSTR can not be used in a math equation. If the SearchCriteria is of type BOOLEAN then EVAL will return TRUE or FALSE.

10.2 Additional Routines

Function GETFONT(TextHandle : HANDLE) : INTEGER;

Function GETFONT returns the ID number of the font in a text block that is connected to TextHandle.

Function GETSIZE (TextHandle : HANDLE) : INTEGER;

Function GETSIZE returns the text size in points of the text block that is connected to TextHandle.

Function GETSTYLE(TextHandle : HANDLE) : INTEGER;

Function GETSTYLE returns the text style of the text block that is connected to TextHandle.

0 = Plain

2 = Bold

4 = *Italic*

8 = Underline

16 = Outline

32 = Shadowed

If GETSTYLE returns 6 then text is **Bold** and *Italic* or **TEXT**.

Function GETLVIS(LayerHandle : HANDLE) : INTEGER;

Function GETLVIS returns the layer visibility of the layer that is connected to LayerHandle.

- 0 = Normal
- 1 = Grayed
- 2 = Invisible

Procedure HMOVE(ObjectHandle : HANDLE; XOffset, YOffset : REAL);

Procedure HMOVE moves an object connected by ObjectHandle by a X and Y offset. (Relative move)

PROCEDURE SetActSymbol(s:string);

Sets symbol (s:string) to be active. Same as selecting symbol from library.

FUNCTION ActSymDef:Handle;

Returns a handle to the currently active symbol.

FUNCTION LNewObject:Handle;

Returns a handle to the last object created from Minipascal. It returns nil if no objects have been created by Minipascal in this session.

PROCEDURE SetTool(thetool:integer);

Sets the current tool from the palette;

The Hand	= 1;	Round Rect	= 9;
The Cursor	= 2;	Oval	= 10;
Zoom In	= 3;	Quarter Arc	= 11;
Zoom Out	= 4;	Arc	= 12;
Constrained Line	= 5;	Curve	= 13;
Line	= 6;	Polygon	= 14;
Text	= 7;	Locus	= 15;
Rect	= 8;	Symbol	= 16;

For Alternates, add the following:

Alternate 1 = 256

Alternate 2 = 512

Alternate 3 = 1024

Alternate 4 = 2048

ex. SetTool(Arc+256) sets the Arc by three points tool.

PROCEDURE SetConstrain(s:string);

ex: Setconstrain('AQW');

{ Sets the grid, object and surface snap modes}

Set the current constrain mode. The routine take a string of characters from the keyboard commands for the constrain modes.

A = Grid Snap

Q = Object Snap

S = Surface Snap

W = Intersection Snap

D = Snap along line

E = Parallel Constrain

F = Perpendicular Constrain

R = Angle Constrain

G = Symetrical Constrain

T = Tangent Constrain

Glossary

Absolute method:	the method of point designation in which the coordinates provided represent the actual coordinate locations displayed on the screen.
Actual parameter, argument:	two phrases that refer to the value or variable actually passed to a subroutine.
Assignment statement:	a statement that gives a value to a variable or function.
Block:	the declaration and statement parts of a subroutine. The scope of identifiers is limited to the block they are defined or declared in, and to blocks created within that block.
Comment:	an explanatory note about subprogram operation that is ignored by the compiler.
Compatible:	two variables or expressions are type-compatible if they represent values of the same type.
Compile:	converting a subprogram from English into a code that the computer can actually execute.
Compile-time error:	a bug that is caught by the compiler and must be fixed before the subprogram can execute properly.
Compound statement:	a series of statements between a BEGIN and END that form a unit and are treated semantically like a single statement.
Concatenate:	the combining of two strings to form one.
Conditional statement:	statements that determine decisions.
Control statement:	statements that control the execution of an action.
Distance-Angle method:	the point-designation method of using a distance and an angle value to represent a location.
DIV:	a reserved word and operator which returns the quotient of a division.
Definition part:	the segment of a subprogram in which labels and constants are defined.

Delimiter:	a word or symbol that marks a boundary for the compiler.
Execute:	to carry out a statement or a series of statements.
Expression:	any representation of a value in MiniPascal.
Formal parameter:	parameters located in the declaration section of a subprogram.
Function declaration:	the actions of a function.
Global identifier:	identifiers which are created in the main subprogram.
Gradians:	a unit of measure of angles. $100 \text{ gradians} = 90^\circ$.
Identifier:	a word whose meaning is defined by the programmer.
Local identifier:	identifiers which are created in an internal subroutine.
MOD:	a reserved word and operator which returns the remainder from a division.
Nesting:	a statement that is the action of a similar type of statement.
Operand, Operator:	an operator is a symbol (such as '+' or '*') which joins with operands (representation of values) to form expressions.
Operator hierarchy:	a scheme that determines the order in which operations in an expression are carried out.
Parameter:	a variable that is used to transport information between subprograms.
Parameter list:	the portion of an internal subroutine in which value- and variable-parameters are declared.
Precedence:	the notion of operator precedence lets rules be established for determining the order in which expressions are evaluated.
Pre-defined identifier (special word):	a constant that is accessible without being defined or declared by the programmer.
Procedure:	a subroutine which handles a specific task of a main subprogram.
Procedure call:	the invocation of a procedure.
Procedure declaration:	the statements of an internal subroutine.
Programmer:	the person using MiniPascal in order to create macros.

Quoted character constant:	a string which only has one character.
Quoted string constants (strings):	a sequence of characters.
Radian:	a unit of measure of angles. 2π radians equals 360° .
Relative method:	the point-designation method of providing a horizontal and vertical movement from the current coordinate location.
Repetition statement:	statements that are capable of performing repeated actions.
Reserved word:	a word which is part of the basic vocabulary of MiniPascal.
Run-time error:	a mistake that occurs during execution of a subprogram.
Scalar types:	a data type which includes <i>integer</i> , <i>longint</i> , <i>boolean</i> and <i>char</i> types.
Scope:	the scope of an identifier is its range of meaning within a subprogram.
Simple types:	the standard MiniPascal types, these include <i>real</i> , <i>string</i> and scalar types.
Standard function, standard procedure:	procedures and functions that are pre-defined in MiniPascal.
Statement:	MiniPascal's unit of activity.
String (quoted string constant):	a series of characters.
Structured type:	structured types provide means of storing information.
Subprogram:	a procedure or a function in MiniPascal, similar to a subroutine.
Undefined:	a variable or function that has not been assigned a value.
Undeclared:	an identifier which has not been declared or defined within a subprogram and is not a reserved or pre-defined word.
User:	the person using a written macro.
Variables:	a means of storing different types of information.

Appendix A

Solutions to Exercises

Unit I Exercises

1. Use the RECT procedure to create three rectangles of equal size and align them side by side.

Answer:

(Answers may differ)

```
RECT(-1,1,0,0);
```

```
RECT(0,1,1,0);
```

```
RECT(1,1,2,0);
```

2. Use the OVAL procedure to create a circle.

Answer:

(Answers may differ)

```
OVAL(0,1,1,0);
```

3. Use the RRECT procedure and the Distance-Angle method to create a rounded rectangle.

Answer:

(Answers may differ)

```
MOVETO(0,0);
```

```
RRECT(1",#90,1",#0,0.5,0.5);
```

4. Use the BEINGROUP and ENDGROUP procedures to group an oval and a rectangle.

Answer:

(Answers may differ)

```
BEGINGROUP;
```

```
    OVAL(0,1,1,0);
```

```
    RECT(1,1,2,0);
```

```
ENDGROUP;
```

5. Use the POLY procedure to create an octagon.

Answer:

(Answers may differ)

ClosePoly;

Poly(

-775/1024",25/1024",

249/1024",25/1024",

973/1024",749/1024",

973/1024",1 749/1024",

249/1024",2 449/1024",

-775/1024",2 449/1024",

-1 475/1024",1 749/1024",

-1 475/1024",749/1024");

Unit II Exercises:

Name the variable type of each example below:

1. 100,000

Answer: *Longint*

2. FALSE

Answer: *Boolean*

3. 'K'

Answer: *String* or *Char*

4. 'England'

Answer: *String*

5. 1.45

Answer: *Real*

Appendix B

Reserved Words and the ASCII Character Set

MiniPascal Reserved Words:

AND	DO	FUNCTION	MOD
THEN	WHILE	BEGIN	ELSE
GOTO	NOT	TO	CONST
END	IF	PROCEDURE	UNTIL
DIV	FOR	LABEL	REPEAT
VAR	ARRAY	OF	

ASCII Character Set:

Each table entry contains a character and its corresponding ASCII code.

EOT	CR	SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/	0
4	13	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	@	A	B	C
49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67
D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86
W	X	Y	Z	[\]	^	_	`	a	b	c	d	e	f	g	h	i
87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105
j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{	
106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124
}	~	del	A	A	C	E	H	O	U	e	e	e	e	e	e	e	e	e
125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
e	e	i	i	i	i	n	o	o	o	o	o	u	u	u	u	r	o	e
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160	161	162
E	S	e	q	B	e	e	=	.	-	*	E	B	o	±	£	¿	¥	μ
163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180	181
ø	Σ	Π	π		g	g	Ω	æ	ß	ç	í	¬	✓	†	≈	Δ	«	»
182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200
...	™	À	Á	Â	Ã	Ä	Å	—	—	“	”	’	÷	◊	ü			
201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216			

— stands for a nonbreaking space, the same width as a digit.

Notes:

Index

A	
abs	199
absolute	20, 134, 239
actlayer	207
actssheet	207
actual parameter	70, 239
addpoint	115
alrtdialog	161
and	
precedence of	41 - 43, 45
ang2vec	232
angbvec	232
angdialog	154
angdialog3D	161
angle	190
anglevar	135
append	169
arc	81
arccos	200
arcsin	200
arctan	200
area	190
argument	70, 74
arithmetic operators	37 - 41
array	28, 227
arrowhead	117
arrowsize	118
ASCII	244
assignment	
operator	38, 53 - 56
statement	53 - 56, 239
autokey	237
B	
backward	146
begin	33 - 34, 58
beginfolder..endfolder	106
begingroup..endgroup	94 - 98
beginmesh..endmesh	107
beginmxtrd..endmxtrd	109
beginpoly..endpoly	115
beginsweep..endsweep	112
beginsym..endsym	105
begintext..endtext	107
beginxtrd..endxtrd	109 - 112

block	72, 239
boolean	
expressions	41 - 45, 54 - 55
operators	41
botbound	196

C	
capslock	238
cellhasnum	215
cellhasstr	215
cellstring	189
cellvalue	189
char	27 - 28
chr	201
classlist	219
classnum	219
close	72, 170
closepoly	118
clrmenu	175
clrmessage	116
command	238
comments	19, 36
comp	232
compatible types	28 - 29, 239
compiler	28
compound statement	58, 239
concat	202, 2439
const	52
constants	51 - 52
control structures	
for..to,downto	62 - 63
goto	66
if..then..else	57, 59 - 60
repeat..until	64
while do	65
copy	202
copymode	119
cos	199
count	194

D	
data types	27
boolean	27
char	28
integer	27
longint	27
real	28
text	28

date	150
declarations	51 - 53
function	70, 240
procedure	69, 240
variable	51
definition parts	52, 71, 239
constants	51 - 53
label	66
deg2rad	200
delclass	151
delete	202
deleteobjs	144
delname	151
delimiter	36, 240
didcancel	164
dimarctext	8
dimtext	82
distance	204
distdialog	155
div	34, 39, 239
domenu	176
doublines	136
downto	62
drwsize	139
dselectall	80
dselectobj	193
duplicate	144

E else	57 59
end	33 - 34, 58
eof	168
eoln	168
equalpt	204
equalrect	204
execute	240
exp	199
expressions	37 - 50, 240
arithmetic	37 - 41
boolean	41 - 45
evaluation of	37 - 50

F factlayer	206
farrowhead	219
ffillback	220
ffillfore	220
ffillpat	219

files	
text	15, 77 - 78
fillback	122
fillfore	122
fillpat	121
fin3d	208
finfolder	209
fingroup	208
finlayer	209
finsymdef	208
flayer	207
fliphor	145
flipver	145
fnderror	152
fobject	206
for	62
forward	146
fpenback	220
fpenfore	220
fpenpat	219
fpen size	219
fsactlayer	207
fsubject	207
fsymdef	208
function	70
function	
assignment to	71
heading	70 - 71

G	
get3dcntr	214
get3dinfo	215
getarc	214
getbbox	211
getcalign	216
getcellnum	215
getcellstr	215
getclass	211
getcwidth	215
getdimtext	213
getfile	165
getfname	218
getfpat	212
getkeydown	235
getlayer	208
getline	233
getlname	217

getlocpt	21
getls	212
getlscale	217
getlw	212
getmouse	238
getname	211
getobject	207
getorigin	219
getpolypt	214
getpt	234
getptl	234
getrect	233
getrrdiam	214
getsdfld1	218
getsdfld2	218
getsdfld3	218
getsdfld4	218
getsdname	217
getsegpt1	213
getsegpt2	213
getsymfld1	213
getsymfld2	213
getsymfld3	213
getsymfld4	213
getsymname	213
getsymrot	214
gettext	212
gettxtnum	213
gettype	211
getvertnum	214
global identifiers	72, 240
goto	66
graylayer	126
gridlines	138
group	98

H handle	28, 205
hangle	217
harea	216
hasdim	213
hcenter	217
heading	69 - 71
height	191
hheight	216
hideclass	126
hidelayr	126

	hlength	216
	hperim	216
	hwidth	216
I	if structure	57 - 61
	nested	60
	insert	203
	intdialog	156
	integer	27
K	keydown	236
L	label	35, 66 - 67
	layer	83
	lckobjs	122
	leftbound	195
	len	202
	length	191
	line	84
	lineto	84
	llayer	207
	ln	199
	loadcell	102
	lobjct	206
	local identifiers	72 240
	locus	85
	loop structures	
	for	62 - 64
	repeat	64
	while	65
	lsactlayer	207
M	menuitem	175
	message	116
	mod	240
	mousedown	236
	move	86
	move3d	148
	move3dobj	223
	moveback	146
	movefront	146
	moveobjs	147
	moveto	86

N	nameclass	126
	namelist	219
	namenum	219
	nameobject	120
	nextdobj	225
	nextlayer	225
	nextobj	225
	nextsobj	225
	nextsymdef	226
	nfillclass	126
	noanglevar	135
	norm	232
	not	41, 45
	num2str	201
	num2strf	207
	numlayers	218
	numobj	217
	numsobj	217
O	objecttype	192
	open	165
	openpoly	118
	operand	37, 240
	operators	37 - 48, 240
	boolean	41 - 45
	integer	37
	precedence of	
	see precedence	
	real	37
	relational	45 - 47
	option	238
	or	43 - 44
	precedence of	40 - 41
	ord	201
	oval	87
P	parameter	74, 240
	actual	70
	argument	70, 74
	value	74
	variable	74
	parentheses	33
	in expressions	41
	penback	122

penfore	122
pengrid	138
penloc	88
penpat	122
pensize	123
perim	193
perp	232
pickobject	209
poly	88
poly3d	89
pos	202
precedence	41, 240
of arithmetic operators	41
of boolean operators	41
of relational operators	41
prevdobj	226
prevlayer	225
prevobj	225
prevsobj	225
prevsymdef	226
procedure	240
procedure call	56, 68 - 69
procedures	19, 68
call	56 - 57, 240
heading	71
statement	56
ptdialog	157
ptdialog3d	165
ptinpoly	204
ptinrect	204
putfile	169

R rad2deg	200
radians	241
read	166
readln	167
real	28
realdialog	158
rect	91
redraw	139
relational operators	45 - 48
relative	135, 241
repeat..until	64
reserved word	33 - 34, 241
rewrite	168
rightbound	196

rotate	140
rotate3d	149
round	198
rrect	92 - 93
run	176

S	
scale	150
scope	72, 241
scientific notation	35
sdtxtnum	218
selectall	79
selected	212
selectobj	193
selectss	223
semicolon	20
with if..then..else	57
as statement separator	20
set3dinfo	222
set3drot	222
setarc	222
setbbox	221
setclass	220
setcursor	223
setdselect	220
setfillback	222
setfillfore	221
setfpat	220
setls	221
setlw	221
setname	220
setorigin	139
setpenback	221
setpenfore	221
setpolypt	222
setscale	140
setsegpt1	222
setsegpt2	222
setselect	220
settext	223
setunits	141
setview	223
shift	238
showclass	126
showlayer	126
simple types	27 - 28

sin	199
smooth	124 - 125
snap	141
space	174
sprdalgn	98
sprdborder	99
sprdfmat	99 - 101
sprdsheet	199
sprdsiz	215
sprdwidth	101
sqr	199
sqrt	200
statement	19, 53 - 67, 241
assignment	53 - 55
compound	59, 239
str2num	203
strdialog	159
strings	
assignments to	56
output of	173
subprograms	241
functions	68 - 76
procedures	68 - 76
symbol	31 - 36, 105
symdefnum	218
sysbeep	152

T tab	174
text	15, 77 - 78
textface	126
textflip	128
textfont	129
textjust	129 - 130
textorigin	106
textrotate	131
textsize	132
textspace	133
to	62
topbound	195
true	27
trunc	198
type	27 - 28, 241

U ungroup	98
unionrect	204

units	1
unitvec	232
unlockobjs	122
uprstring	203
V	
value-parameter	74
var	
in parameter declaration	74
in variable declaration	51
variable	27, 51, 241
counter	62
declaration	51
global, local	72
scope of	72
variable parameter	7
vdelete	151
vec2ang	232
vrestore	151
vsave	151
W	
wait	153
while..do	65
width	194
write	170 - 173
writeln	173
X	
xcenter	194
Y	
ycenter	194
yndialog	160

MiniPascal Quick Reference Section

Main Subprogram Outline:

PROCEDURE MySubprogram;

{Definition Part}

LABEL 1,2;

CONST

myConst = 5.2424;

{Declaration Part}

VAR

myVariable : INTEGER;

{Internal Procedures or Functions}

{Statement Part}

BEGIN

RECT(0,1,1,0);

{Procedure call}

myVariable := 4 + 2;

{Simple statement}

IF myVariable = 5

{Structured statement}

THEN

BEGIN

{Compound statement}

myVariable {Identifier} := 35 {Operand} + {Operator} - 4;

myVariable := 5 * 4.23; {Expression}

END;

{Control Structures}

FOR myVariable := 1 TO 5 DO {FOR..DO Structure}

WRITELN('myVariable = ',myVariable);

IF myVariable = 3

{IF..THEN..ELSE Structure}

THEN

myVariable := 7

{Note no semicolon prior to ELSE}

ELSE {optional}

myVariable := 8;

WHILE myVariable < 10 DO {WHILE..DO Structure}

WRITELN('myVariable = ',myVariable);

REPEAT {REPEAT..UNTIL Structure}

myVariable := myVariable + 1;

UNTIL myVariable = 12;

END;

RUN(MySubprogram);

Output Format:

WRITELN(AnyValue:FieldWidth);

WRITELN(RealValue:FieldWidth:DecimalPlaces);



Installing MiniCad+ on your Hard Drive

Using the MiniCad+ 3.0 Installer:

1. Create a folder on your Desktop.
2. Insert the MiniCad+ 3.0 Disk 1 (Serial Numbered disk) into the disk drive.
3. Double click on the MiniCad+ 3.0 Installer icon.
4. In the dialog box select your hard drive by pressing the 'Drive' radio button.
5. Select the folder created in Step 1 by double clicking on it.
6. Click on the 'Save' radio button.
7. When that is finished you will be presented with a dialog box that says:
"Please open the file 2.MiniCad+? "
8. Click on the 'Drive' radio button until "1.MiniCad+" is selected.
9. Click on the 'Eject' radio button to eject Disk 1.
10. Insert Disk 2 into the disk drive.
11. Click on the 'Open' radio button.
12. When that is complete you should be presented with a dialog box saying:
" All done! MiniCad+ was installed successfully."
13. Click on the 'OK' radio button.
14. MiniCad+ 3.0 is now residing in the folder created in Step 1.
15. ENJOY!